



THE COMPLETE GUIDE TO RAILS PERFORMANCE

NATE BERKOPEC

Table of Contents

| | |
|--|-------|
| Introduction | 1.1 |
| Setting Up Rubygems.org | 1.2 |
| Principles and Tools | 1.3 |
| An Economist, A Physicist, and a Linguist Walk Into a Bar... | 1.3.1 |
| Little's Law | 1.3.2 |
| The Business Case for Performance | 1.3.3 |
| Performance Testing | 1.3.4 |
| Profiling | 1.3.5 |
| Memory | 1.3.6 |
| Rack Mini Profiler | 1.3.7 |
| New Relic | 1.3.8 |
| Skylight | 1.3.9 |
| Optimizing the Front-end | 1.4 |
| Chrome Timeline | 1.4.1 |
| The Optimal Head Tag | 1.4.2 |
| Resource Hints | 1.4.3 |
| Turbolinks and View-Over-The-Wire | 1.4.4 |
| Webfonts | 1.4.5 |
| HTTP/2 | 1.4.6 |
| JavaScript | 1.4.7 |
| HTTP Caching | 1.4.8 |
| Optimizing Ruby | 1.5 |
| Memory Bloat | 1.5.1 |
| Memory Leaks | 1.5.2 |
| ActiveRecord | 1.5.3 |
| Background Jobs | 1.5.4 |
| Caching | 1.5.5 |

| | |
|------------------------------|--------|
| Slimming Down Your Framework | 1.5.6 |
| Exceptions as Flow Control | 1.5.7 |
| Webserver Choice | 1.5.8 |
| Idioms | 1.5.9 |
| Streaming | 1.5.10 |
| ActionCable | 1.5.11 |
| The Environment | 1.6 |
| CDNs | 1.6.1 |
| Databases | 1.6.2 |
| JRuby | 1.6.3 |
| Memory Allocators | 1.6.4 |
| SSL | 1.6.5 |
| Easy Mode Stack | 1.7 |
| The Complete Checklist | 1.8 |

The Complete Guide to Rails Performance

This course will teach you how to make your Ruby and Rails web applications as fast as possible. Whether or not you're an experienced programmer, this book is intended for anyone running Ruby web applications in production that wants to make them faster.

If you've ever put a Ruby web application in production, you've probably dealt with performance issues. It's the nature of the language - it isn't the fastest on the block. Each line of Ruby code we write tends to make our application just that little bit slower. Some would take this to mean it's time to jump ship and head for a faster language, like Go or Crystal.

Veterans probably already know the futility of chasing the latest programming craze. We all remember the furor over Scala in 2008, when Twitter reportedly dumped their Rails app for a Scala-powered backend. Where are all the Scala applications now? If you had rewritten your app in Scala in 2008, where would you be today?

Plus, I think a lot of us just love writing Ruby. I know I do. Rather than seeing Ruby's native performance as an immovable barrier, I see it as a signal that, as Rubyists, we need to be even more knowledgeable about performance than our counterparts in languages like Go or even JavaScript.

This course was written to help companies and programmers continue to use Ruby for their web applications for many years to come. I love the Ruby programming language - ever since I read *_why's Poignant Guide*, I have been deeply in love with this quirky, beautiful little language. I hope that, by teaching all I know about speed and performance in Ruby and Rails web applications, no one will ever have to feel compelled to choose another language because Ruby could not meet their performance specifications.

Why Rails?

I originally wanted to call this the "Complete Guide to *Ruby and Rails Performance*". Although the "microservices" approach is becoming more popular, and frameworks like Lotus and Sinatra have received more attention, Rails remains the king in the Ruby web framework space. Barring catastrophe, it probably always will be.

In addition, the stated purpose of Rails is quite similar to the goals I had in preparing this course. At RailsConf in 2015, DHH said that he imagined Rails as a kind of "doomsday prepper backpack" - he wanted everything in there that he would need to reconstruct a top-1,000 website with a small team of people. This is, of course, exactly what he has done with Rails at Basecamp - three times, in fact, since Basecamp has been rewritten twice.

However, Rails doesn't come with an instruction manual. There are a lot of performance pitfalls in Rails, and none of them are covered in Rails' documentation (they probably shouldn't be, anyway). This course is that instruction manual. An alternate title might have been "The Doomsday Prepper Backpack for Rails Performance".

That said, much of this book isn't Rails-specific. An entire module isn't even Ruby-specific.

This course is about the entire stack - from the metal to the browser. Anything that can impact end-user experience is covered, from HTML down to SQL. Much of that isn't framework-specific, though some of it will be.

This course is the tool I wish I would have had in my career - a step-by-step instruction and checklist for building a lightning-fast Rails site, geared towards small to medium sized teams and websites. This is where I've spent my entire career, both in full-time and consulting work.

Why You?

This course assumes about 3 months of Rails experience, no more. I hate technical writing that assumes the reader is some kind of genius and doesn't explain (or even just link to an explanation) everything that's going on. In addition, even if you're not *completely* sure you've understood a topic, you can ask me and your fellow participants on our private Slack channel. Finally, if you buy the course and decide its over your head, I'll refund your money. No questions asked.

If you (or your customers) are not satisfied with the speed of your Rails application, this course will work for you. Not 100% will apply, of course, but 90% of it will.

This course is applicable to both "greenfield" and "legacy" applications, but I might say it focuses on legacy applications. I've worked on a lot of what I consider "legacy" applications (2+ year codebases). Those are the ones that tend to be slow, not the

greenfield ones. My only caveat is I'm not going to talk about optimizing previous major versions of anything - Rails 3, Ruby 1.9, etc.

I'm going to focus the course on the typical Rails stack. In my opinion, that includes a SQL relational database. NoSQL is too far outside of my comfort area to speak meaningfully about it. I include a specific section on Postgres, because it has several unique features and it's so widely used. JS frameworks are also not covered specifically (I won't tell you how to optimize React, for example), but I will cover the specific needs of an API-only application.

How To Use This Course

This course is delivered to you in the form of a git repository. I considered using more traditional "MOOC" software, but I didn't like anything that I could find. We're all programmers, so I hope that interacting with this course in a git repository isn't too novel for any of us.

Here is how I suggest you work through this course:

- If you haven't already, join the Complete Guide's official Slack channel. You received an email with an invitation after your purchase. Once you do, you'll be able to get access to the private GitHub organization and git remote for this course. Using that git remote, you'll be able to simply "git pull" to receive the latest course updates and revisions.
- Read the lessons more or less in order. I've provided many ways to do this - HTML, PDF, e-Reader formats, an audio recording, and even JSON. You can even just read this course in your text editor if you like - the folders have been alphabetized to match the order of the Table of Contents, and all lessons are in Markdown format. This course was produced using Gitbook, but installation is not required to view the material.
- *If you purchased the Web-Scale package*, after you've completed a lesson, watch the included screencast for that lesson. The screencasts are included in a separate archive, though they can be merged with your course folders (because the video folder structure is the same). The screencast is *additional* material, not just a restatement of the lesson. Usually, the screencast is a "watch-over-my-shoulder"-style presentation where you'll watch me implement the lesson's concepts.
- Finally, try your hand at the lab for the lesson. Labs are hands-on exercises that test your skills using what you learned in the lesson.

Setting Up Rubygems.org

Rubygems.org is a Rails application that hosts Rubygems for the entire Ruby community. It has a web interface that provides search and several other features. This course uses Rubygems.org as an example application for many of the labs and hands-on exercises in the course.

Here's what you'll need to do to get Rubygems.org running locally:

1. Install Ruby 2.2.3 via your preferred method. [I use `ruby-install`](#) .
2. `git clone https://github.com/rubygems/rubygems.org.git`
3. This lab requires that you check out a specific commit. `git checkout e0009000`
4. Rubygems.org requires a working installation of Postgres and Redis. Install these tools if you haven't already.
5. `cp config/database.yml.example config/database.yml` and modify as required.
Create the production and development databases: `bundle exec rake db:reset` and `RAILS_ENV=production bundle exec rake db:reset` .
6. Run `bundle install` and make sure you have Postgres and Redis running.

At this point, you should have a working copy - at least in development mode. Trying start a server in development mode and make sure it works.

To get Rubygems.org running in production mode, follow these steps:

1. [Download a copy of Rubygems.org's production database](#) and load it into your production database (`gemcutter_production`). This copy of the Rubygems.org production database is from the same date as the commit we've checked out, and sanitized of any sensitive information. This dump can be loaded into Postgres with `$ psql gemcutter_production < PostgreSQL.sql` . More recent dumps are available [here](#).
2. [Download a copy of Rubygems.org's production Redis dump](#) (also from the same date as the commit we've checked out). Extract this dump, rename it to `dump.rdb` and place it in the root of the application - now, using `redis-server` from the root of your application will automatically load this database dump. Additional, more recent Redis dumps are available [here](#).
3. In `config/environments/production.rb` , change `config.force_ssl` to `false` and `config.serve_static_files` to `true` .
4. In `config/application.rb` , delete or comment out the line that says

```
config.middleware.use "Redirector" unless Rails.env.development? .
```

5. Generate the assets: `RAILS_ENV=production rake assets:precompile`

6. Start a production server with `RAILS_ENV=production SECRET_KEY_BASE=foo rails s`

For more about using Rubygems.org, see their [CONTRIBUTING.MD](#).

Module 1: Principles

This module is about the principles of Rails performance. We'll be covering the "bigger picture" in these chapters, and we'll also talk about some of the more general tools, like profilers and benchmarking, that we can use to solve performance problems.

The most important lesson in this module is on **The 80/20 Principle**. This principle is like an "Occam's Razor" against the evil of premature optimization. Without an acknowledgement that performance problems are often limited to just a few small areas of our application, we may be tempted to blindly optimize and tweak areas of our application that aren't even the bottleneck for our end-users.

An Economist, A Physicist, and a Linguist Walk Into a Bar...

You bought this course because you wanted to learn more about web application performance. Before we get to that, I want to lay out some basic principles - some guiding lights for our future work together. Actually, I want to tell you about a physicist from Schenectady, a Harvard linguist, and an Italian economist.

The Italian economist you may already have heard of - Vilifred Pareto. He became famous for something called **The Pareto Principle**, something you might be familiar with. So why am I spending an entire lesson on it? Because while you've probably *heard* of the Pareto Principle, I want you to *understand why* it actually works. And to do that, we're going to have to look back in history.

Benford - the physicist

Frank Benford was an American electrical engineer and physicist who worked for General Electric. It was the early 20th century, when you had a job for life rather than a startup gig for 18 months, so he worked there from the day he graduated from the University of Michigan until his death 38 years later in 1948. Back in that day, before calculators, if you wanted to know the logarithm of a number - say, 12 - you looked it up in a book. The books were usually organized by the leading digit, so if you wanted to know the logarithm of 330, you first went to the section for 3, then looked for 330. Bedford noticed that the first pages of the book were far more worn out than the last pages. Benford realized this meant that the numbers looked up in the table began more often with 1 than with 9.

Most people would have noticed that and thought nothing of it. But Benford pooled 20,000 numbers from widely divergent sources (he used the numbers in newspaper stories) and found that the leading digit of all those numbers followed a logarithmic distribution too!

This became known as Benford's Law. Here are some other sets of numbers that conform to this logarithmic distribution:

- Physical constants of the universe (pi, the molar constant, etc.)
- Surface areas of rivers

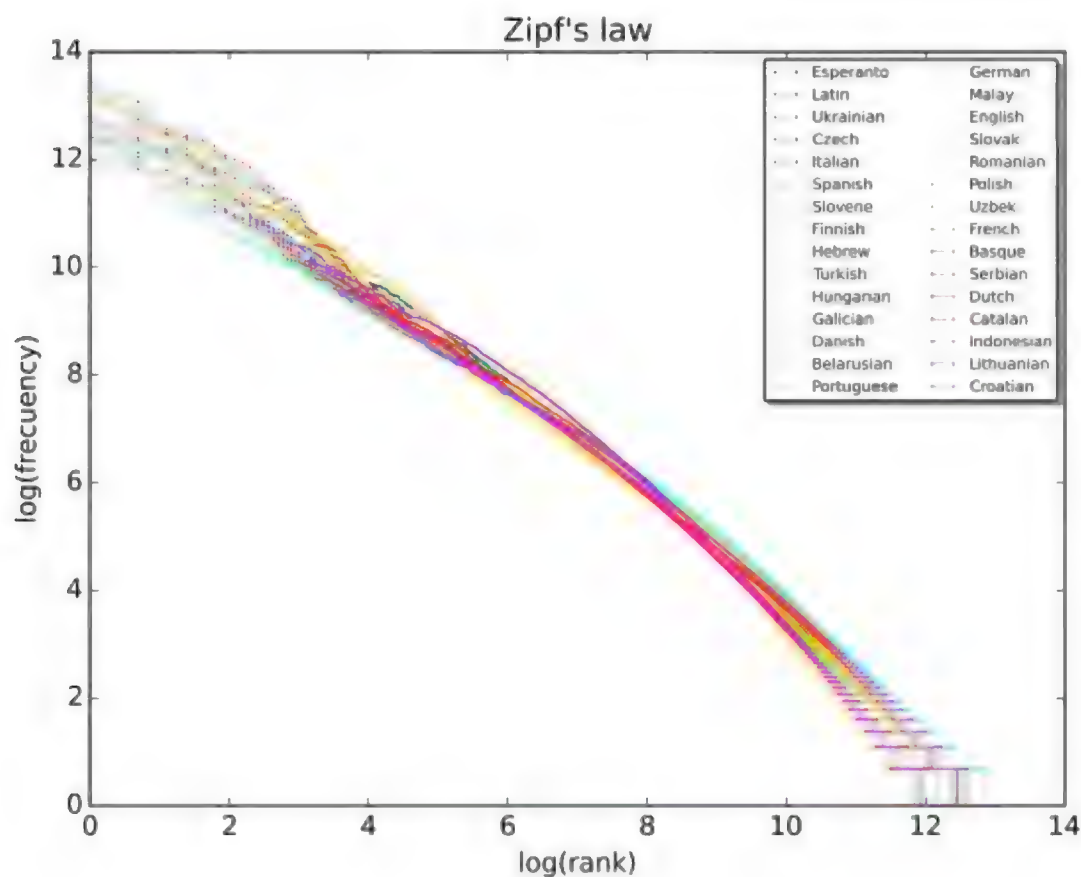
- Fibonacci numbers
- Powers of 2
- Death rates
- Population censuses

Bedford's Law is so airtight that it's been admitted in US courts as evidence of accounting fraud (someone used RAND in their Excel sheet!). It's been used to identify other types of fraud too - elections and even macroeconomic data.

What would cause numbers that have (seemingly) little relationship with each other to conform so perfectly to this non-random distribution?

Zipf - the linguist

At almost exactly the same time, George Kingsley Zipf was studying languages at Harvard. Uniquely, George was applying the techniques of a new and interesting field - statistics - to the study of language. This landed him an astonishing insight: in nearly every language, some words are used a lot, but most (nearly all) words are used hardly at all. That is to say, if you took every English word ever written and plotted the frequency of words used as a histogram, you'd end up with a graph that looked something like what you see below - a plot of the rank versus frequency for the first 10 million words in 30 different languages of Wikipedia. Note the logarithmic scales.

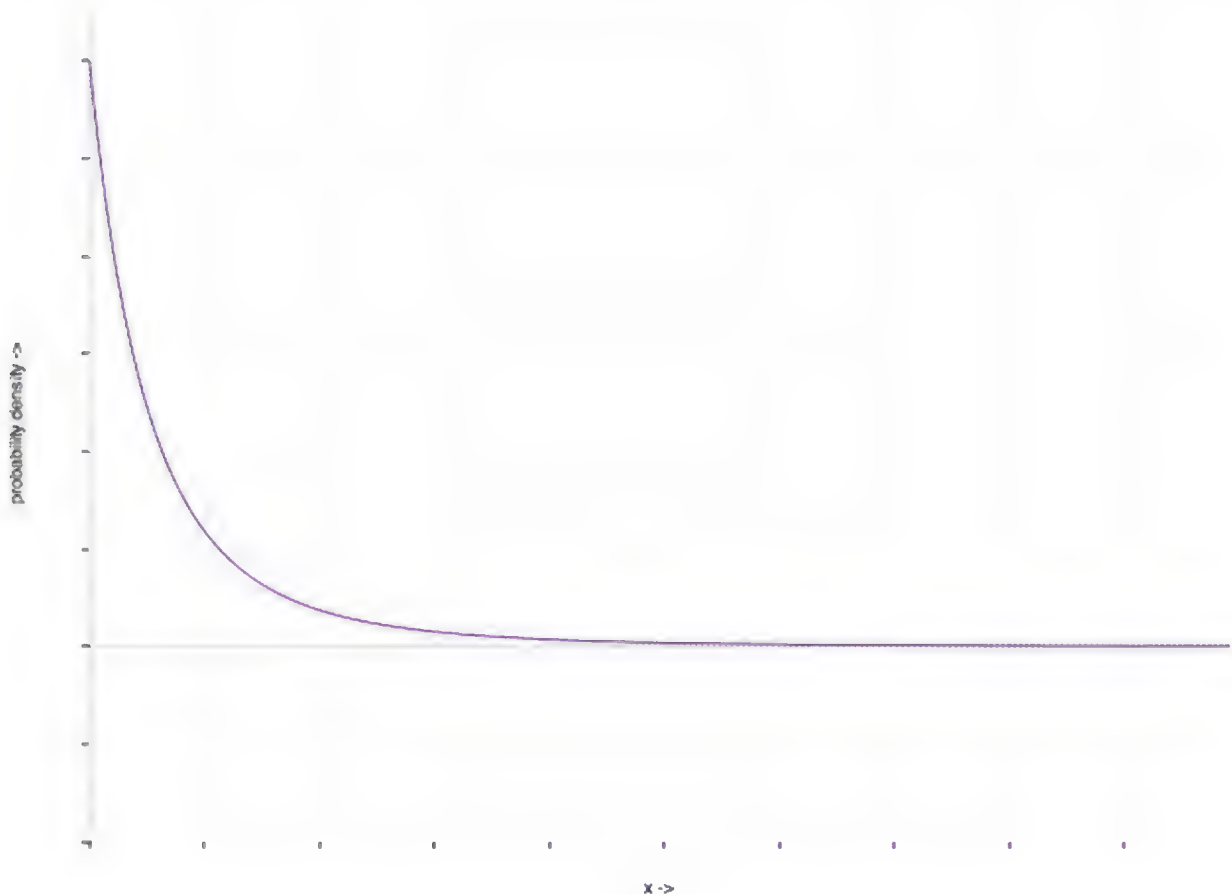


The Brown Corpus is 500 samples of English-language text comprising 1 million words. Just 135 unique words are needed to account for 50% of those million. That's insane.

If you take Zipf's probability distribution and make it continuous instead of discrete, you get the Pareto distribution.

Pareto - the economist

Pareto initially noticed this distribution when he was thinking about wealth in society - he noticed that 80% of the wealth and income came from 20% of the people in it.



The Pareto distribution, pictured, has been found to hold for a scary number of completely different and unrelated fields in the sciences. For example, here are some natural phenomena that exhibit a Pareto (power law) distribution:

- Wealth inequality
- Sizes of rocks on a beach
- Hard disk drive error rates (!)
- File size distribution of Internet traffic (!!!)

We tend to think of the natural world as random or chaotic. In schools, we're taught the bell curve/normal distribution. But reality isn't normally distributed. It's log-normal. Many probability distributions, in the wild, support the Pareto Principle:

80% of the output will come from 20% of the input

While you may have heard this before, what I'm trying to get across to you is that it isn't made up. The Pareto distribution is used in hundreds of otherwise completely unrelated scientific fields - and we can use its ubiquity to our advantage.

It doesn't matter what area you're working in - if you're applying equal effort to all areas, you *are wasting your time*. What the Pareto distribution shows us is that most of the time, our efforts would be better spent *finding* and *identifying* the crucial 20% that

accounts for 80% of the output.

Allow me to reformulate and apply this to web application performance:

80% of an application's work occurs in 20% of its code.

There are other applications in our performance realm too:

80% of an application's traffic will come from 20% of its features.

80% of an application's memory usage will come from 20% of its allocated objects.

The ratio isn't always 80/20. Usually it's way more severe - 90/10, 95/5, 99/1. Sometimes it's less severe. So long as it isn't 50/50 we're talking about a non-normal distribution.

This is why premature optimization is so bad and why performance monitoring, profiling and benchmarking are so important. What the Pareto Principle reveals to us is that optimizing any random line of code in our application is in fact unlikely to speed up our application at all! 80% of the "slowness" in any given app will be hidden away in a minority of the code. So instead of optimizing blindly, applying principles at random that we read from blog posts, or engaging in Hacker-News-Driven-Development by using the latest and "most performant" web technologies, we need to measure where the bottlenecks and problem areas are in our application.

Repeat after me: I will not optimize anything in my application until my metrics tell me so.

There's only one skill in this entire course that you need to understand completely and deeply - how to *measure* your application's performance. Once you have that skill mastered, reading *every single lesson* in this course might be a waste of time. Your problems are not other's problems. There are going to be lessons in this course that solve problems you don't have (or don't comprise that crucial 20% of the causes of slowness in your application). For that reason, with the exception of this first measurement module, I do *not* encourage you to read this entire course from cover to cover. Complete the first module, then use the skills you learn there to measure the bottlenecks in your application. From there, find the lessons that apply to your current performance issues.

On the flip side, you should realize that the Pareto Principle is extremely liberating. You *don't* need to fix every performance issue in your application. You don't need to go line-by-line to look for problems under every rock. You need to *measure* the actual performance of your application, and focus on the 20% of your code that is the worst performance offender.

To beat this into your skull even further, and to give you a preview of some future material, I'm going to tell you a story.

My talk for Rubyconf 2015 was a guided read through of Minitest, the Ruby testing framework. It's a great read if you've got a spare hour or two - it's fairly short and quite readable. As I was reading Minitest's code myself, I came across this funny line:

```
def self.runnable_methods
  methods = methods_matching(/^test_/)

  case self.test_order
  when :random, :parallel then
    max = methods.size
    methods.sort.sort_by { rand max }
  when :alpha, :sorted then
    methods.sort
  else
    raise "Unknown test_order: #{self.test_order.inspect}"
  end
end
```

This code is extremely readable as to what's going on; we determine which methods on a class are runnable with a regex ("starts with test_"), and then sort them depending upon this class's `test_order`. Minitest uses the return value to execute all of the `runnable_methods` on all the test classes you give it. Usually this is a randomized array of method names.

What I was honing in on was this line, which is run when `:test_order` is `:random` or `:parallel` (Minitest's default):

```
max = methods.size
methods.sort.sort_by { rand max }
```

This seemed like a really roundabout way to do `methods.shuffle` to me. Maybe Ryan (Minitest's author) was doing some weird thing to ensure deterministic execution given a seed (Minitest runs your tests in the same order given the same seed to the random number generator. It turns out `methods.shuffle` is deterministic, though, just like the code as written). I decided to benchmark it, mostly out of curiosity.

Whenever I need to write a micro benchmark of Ruby code, I reach for [benchmark/ips](#)¹. `ips` stands for iterations-per-second. The gem is an extension of the `Benchmark` module, something we get in the Ruby stdlib.

Sidenote:

Here's that benchmark:

```
require "benchmark/ips"

class TestBench
  def methods
    @methods ||= ("a".."z").to_a
  end

  def fast
    methods.shuffle
  end

  def slow
    max = methods.size
    methods.sort.sort_by { rand max }
  end
end

test = TestBench.new

Benchmark.ips do |x|
  x.report("faster alternative") { test.fast }
  x.report("current minitest code") { test.slow }
  x.compare!
end
```

It suggested (as I suspected), that `shuffle` was 12x faster than `sort.sort_by { rand methods.size }`. This makes sense - `shuffle` randomizes the array with C, which will always be faster than randomizing it with pure Ruby. In addition, Ryan was actually sorting the array twice - once in alphabetical order, followed by a random shuffle based on the output of `rand`.

I asked Ryan Davis, `minitest` author, what was up with this. He gave me a great reply: "you benchmarked it, but did you profile it?"

What did he mean by this? Well, first, you have to know the difference between benchmarking and profiling.

There are a lot of different ways to define this difference. Here's my attempt:

- **Benchmarking.** When we benchmark, we take two competing pieces of code - could be as simple as a one liner, like here, or as complex as an entire web framework. Then, we put them up against each other (usually in terms of

iterations/second) on a simple, contrived task. At the end of the task, we come up with a single metric - a score. We use the score to compare the two competing options. In my example above, it was just how fast each line could shuffle an array. If you were benchmarking web frameworks, you might test how fast a framework can return a simple "Hello World" response. The gist: benchmarks put the competing alternatives on exactly equal footing by coming up with a contrived, simple, non-real-world example. It's usually too difficult to benchmark real-world code because the alternatives aren't doing *exactly* the same thing.

- **Profiling.** When we profile, we're usually examining the performance characteristics of an entire, real-world application. For example, this might be a web application or a test suite. Because profiling works with real-world code, we can't really use it to compare competing alternatives, because the alternative usually doesn't exactly match what we're profiling. Profiling doesn't usually produce a comparable "score" at the end with which to measure these alternatives, either. But that's not to say profiling is useless - it can tell us a lot of valuable things, like what percentage of CPU time was used where, where memory was allocated, and things like that.

What Ryan was asking me was - "Yeah, that way is faster on this one line, but does it really matter in the grand scheme of Minitest"?

Is this one line really part of Pareto's "20%"? We can assume, based on the Principle, that 80% of Minitest's wall time will come from just 20% of its code. Was this line part of that 20%?

So I've already shown you how to benchmark on the micro scale. But before we get to profiling, I'm going to do a quick macro-benchmark to test my assumption that using `shuffle` instead of `sort.sort_by` will speed up minitest.

Minitest is used to run tests, so we're going to benchmark a whole test suite. Rubygems.org will make a good example test suite.

When micro-benchmarking, I reach for `benchmark-ips`. When macro-benchmarking (and especially in this case, with a test suite), I usually reach first for the simplest tool available: `time` ! We're going to run the tests 10 times, and then divide the total time by 10.

```
time for i in {1..10}; do bundle exec rake; done

...

real    15m59.384s
user    11m39.100s
sys     1m15.767s
```

When using `time`, we're usually only going to pay attention the `user` figure. `real` gives the actual total time (as if you had used a stopwatch), `sys` gives the time spent in the kernel (in a test run, this would be things like shelling out to I/O), and `user` will be the closest approximation to time actually spent running Ruby. You'll notice that `user` and `sys` don't add up to `real` - the difference is time spent waiting on the CPU while other operations (like running my web browser, etc) block.

With stock `minitest`, the whole thing takes 11 minutes and 39 seconds, for an average of 69.9 seconds per run. Now, let's alter the Gemfile to point to a modified version (with `shuffle` on the line in question) of `minitest` on my local machine:

```
gem 'minitest', require: false, path: '../minitest'
```

To make sure the test is 100% fair, I only make the change to my local version after I check out `minitest` to the same version that Rubygems.org is running (5.8.1).²

The result? 11 minutes 56 seconds.³ Longer than the original test! We know my code is faster in micro, but the macro benchmark told me that it actually takes longer. A lot of things can cause this (the most likely being other stuff running on my machine), but what's clear is this - my little patch doesn't seem to be making a big difference to the big picture of someone's test suite. While making this change *would*, in *theory*, speed up someone's suite, in reality, the impact is so minuscule that it didn't really matter.

Repeat after me: I will not optimize anything in my application until my metrics tell me so.

¹. The reason I use `benchmark/ips` rather than the `stdlib` benchmark is because the `stdlib` version requires you to run a certain line of code X number of times and tells you how long that took. The problem with that is that I don't usually know how fast the code is to begin with, so I have no idea how to set X. Usually I run the code a few times, guess at a number of X that will make the benchmark take 10 seconds to run, and then move on. `benchmark/ips` does that work for me by running my benchmark for 10 seconds and calculating iterations-per-second. ↩

². Since minitest does not ship with a gemspec, I have to add a bogus one myself.

↩

³. There are far better ways to macro-benchmark this code, which we'll get into later in the course. Also, it might benefit us to profile an entire Minitest test run to see the real 80/20 breakdown. All stuff you're going to learn in this module, just not in this lesson. ↩

Little's Law

How many servers do you need?

I usually see applications over-scaled when a developer doesn't understand how many requests their server can process per second. They don't have a sense of "how many requests/minute equals how many servers?"

I already explained a practical way to determine this - measuring and responding to changes in request queueing time. But there's also a theoretical tool we can use - [Little's Law](#).

$$l = \lambda w$$

In our case, l is the number of application instances we need, λ is the average web request arrival rate (e.g. 1000 requests/second), and w is the average response time of your application in seconds.

First off, some definitions - as mentioned above, the application instance is the atomic unit of your setup. Its job is to process a single request independently and send it back to the client. When using Webrick, your application instance is the entire Webrick process. When using Puma in threaded mode, I will define the *entire Puma process* as your application instance when using MRI, and when using JRuby, *each thread* counts as an application instance. When using Unicorn, Puma (clustered) or Passenger, your application instance is *each "worker" process*.

Let's do the math for a typical Rails app, with the prototypical setup - Unicorn. Let's say each Unicorn process forks 3 Unicorn workers. So our single-server app actually has 3 application instances. If this app is getting 1 request per second, and its average server response time is 300ms, it only needs $1 * 0.3 = 0.3$ app instances to service its load. So we're only using 10% of our available server capacity here! What's our application's theoretical maximum capacity? Just change the unknowns:

$$l/w = \lambda$$

l , the number of instances we have, is 3. w , the average response time, is 0.3 seconds. So for our example app, our theoretical maximum throughput is $3 / 0.3$, or 10 requests per second!

But theory is never reality. Unfortunately, Little's Law is only true *in the long run*, meaning that things like a wide, varying distribution of server response times (some requests take 0.1 seconds to process, others 1 second) or a wide distribution of arrival times can make the equation inaccurate. But it's a good "rule of thumb" to think about whether or not you might be over-scaled.¹

Recall again that scaling hosts doesn't directly increase server response times, it can only increase the number of servers available to work on our request queue. If the average number of requests waiting in the queue is less than 1, our servers are not working at 100% capacity and the benefits to scaling hosts are marginal (i.e., not 100%). The maximum benefit is obtained when there is always at least 1 request in the queue. There are probably good reasons to scale *before* that point is reached, especially if you have slow server response times. But you should be aware of the rapidly decreasing marginal returns.

So when setting your host counts, try doing the math with Little's Law. If you're scaling hosts when, according to Little's Law, you're only at 25% or less of your maximum capacity, then you might be scaling prematurely. Alternatively, as mentioned above, spending a large amount of time per-request in the request queue as measured on NewRelic is a good indication that it's time to scale hosts.

Checking the math

In [April 2007](#), a [presentation was given at SDForum Silicon Valley](#) by a Twitter engineer on how they were scaling Twitter. At the time, Twitter was still fully a Rails app. In that presentation, the engineer gave the following numbers:

- 600 requests/second
- 180 application instances (mongrel)
- About 300ms average server response time

So Twitter's theoretical instances required, in 2007, was $600 * 0.3$, or 180! And it appeared that's what they were running. Twitter running at 100% maximum utilization seems like a recipe for disaster - and Twitter did have a lot of scaling issues at the time. It may have been that they were unable to scale to more application instances because they were still stuck with a single database server (yup) and had bottlenecks elsewhere in the system that wouldn't be solved by more instances.

As a more recent example, in 2013 at Big Ruby Shopify engineer John Duff gave a presentation on [How Shopify Scales Rails \(YouTube\)](#). In that presentation, he claimed:

- Shopify receives 833 requests/second.
- They average a 72ms response time
- They run 53 application servers with a total of 1172 application instances (!!!) with NGINX and Unicorn.

Shopify's theoretical required instance count is 833×0.072 just ~60 application instances. So why are they using 1172 and wasting (theoretically) 95% of their capacity? If application instances block each other in _any way*, like when reading data off a socket to receive a request, Little's Law will fail to hold. This is why I don't count Puma threads as an application instance on MRI. Another cause can be CPU or memory utilization - if an application server is maxing out its CPU or memory, its workers cannot all work at full capacity. This blocking of application instances (anything that stops all 1172 application instances from operating at the same time) can cause major deviations from Little's Law.²

Finally, [Envato posted in 2013 about how Rails scales for them](#). Here's some numbers from them:

- Envato receives 115 requests per second
- They run an average of 147ms response time
- [They run 45 app instances](#).

So the math is 115×0.147 , which means Envato theoretically requires ~17 app instances to serve their load. They're running at 37% of their theoretical maximum, which is a good ratio.

Checklist for Your App

- Ensure your application instances conform to a reasonable ratio of what Little's Law says you need to serve your average load.
- Across your application, 95th percentile times should be within a 4:1 ratio of the average time required for a particular controller endpoint.
- No controller endpoint's average response time should be more than 4 times the overall application's average response time.

¹. In addition, think about what these caveats mean for scaling. You can only maximize your actual throughput if requests are as close to the median as possible. An app with a predictable response time is a scalable app. In fact, you may obtain more accurate results from Little's Law if, instead of using *average* server response time, you use your *95th percentile* response time. You're only as good as your slowest responses if your server response times are variable and unpredictable. How do you decrease 95th percentile response times?

Aggressively push work into background processes, like Sidekiq or DelayedJob.

↩

². [There is a distributional form of Little's Law](#) that can help with some of these inaccuracies, but unless you're a math PhD, it's probably out of your reach. ↩

The Business Case for Performance

We've all heard it before - websites in 2016 are bloated, ad-riddled performance nightmares. It takes 10MB of data to render a nearly content-less slideshow article that requires you to click "next" 30 times. I can't answer for the downfall of journalism in the modern era, but I do have some insight to offer in to how our performance problems got this bad.

Consider this talk by Maciej Cegłowski of Pinboard on [The Website Obesity Crisis](#). To summarize his argument:

- Most content on the web is primarily text. Everything else is decoration.
- Page sizes average 2MB, and many popular sites like Twitter and Medium are 1MB. However, the purpose of these sites is to deliver text content, which should only take up a few kB.
- True web performance is not about delivering ads faster - it's about delivering content (what the user came for) without spending too much page weight on things they *don't* want.
- Tech teams don't have the blame here - they usually have their great designs shat over by clients or marketing departments that add fat advertisement or tracking Javascript on top.
- "Minimalism" that hijacks your scroll, forces you to download megabytes of assets, and destroys your privacy purely to read a few sentences of text and see an image is not true minimalism.

I agree with everything there, but I take umbrage with the fourth point - that we, as developers, are mostly blameless in this.

We programmers are born optimizers - if there's a way to do it faster or more efficiently, we'll do it. "It's those darn marketing departments", it's argued. I think it's *our* fault for not fighting back.¹

I once received a bit of advice from a programmer that was moving up into higher management at a Fortune 50 company:

The business people, man - they only understand numbers. If you can justify it with numbers, they'll do whatever you tell them.

The Case for Performance

The reason we're suffering from an anti-performance glut on the web is that technical (I'm including design and programmers here) teams cannot adequately *quantify* and *explain* the costs of sacrificing performance to the "business" side of wherever they work.

There's no doubt that, sometimes, even when the performance costs and benefits are outlined, some businesses will have to choose in favor of bloat. Most advertising-based businesses will necessarily skew towards the bloat-y side of the spectrum. Every team will have different needs.

But most website's performance goes to hell in a handbasket when conversations like *this* occur:

Marketing Person: Hey, can you add this <3rd-party> javascript tracker to our site? Their sales guy says it's really easy and would only be one line of code.

Tech Person: Okay!

What I wish they sounded like is this:

Marketing Person: Hey, can you add this <3rd-party> javascript tracker to our site? Their sales guy says it's really easy and would only be one line of code.

Tech: Sure, I'll try it out and get back to you. *A few hours later* Tech: Hey , I tried adding it our our site. Unfortunately, it increases our page size to 1MB, which is 30% over our budget. If we want to add this, we'll have to call a meeting to discuss it, because we set that budget so that mobile users would have the site load in less than 5 seconds.

Because every new feature almost always implies the execution of additional code, every feature imposes some kind of performance cost. Rather than ignore these costs, we should quantify them.

Technical and product teams need to agree on a shared set of performance goals - these are extremely easy to arrive at, because the research on the link between performance and the bottom line is extremely well documented.

I'm going to summarize the existing research/literature on performance and business performance, but you can find a huge repository of these studies at wpostats.com.

Slowdowns are Outages

We all agree that downtime is a bad thing. Companies get obsessed with “5 nines” of uptime. The reason is obvious - if the website is down, you’re not making money.

But do we think about a temporary *slowdown* in the same way we think about a *total outage*? Isn’t a *slowdown* really a *partial outage*?

Here’s an example I’ve seen - a site uses WebFonts provided by a 3rd party service, such as Typekit. The page has no text content visible until those fonts are loaded. Inevitably, the 3rd party font provider experiences either a total outage or a slowdown, and delays your content loading as a result. Let’s say you have a timeout on your font loading of 3 seconds, and then the browser displays its fallback font. You’ve just slowed your site down by 3 seconds until your 3rd party provider can get back online.

This is not a full outage, and all of your traditional uptime monitors will fail to catch this problem automatically.

TRAC Research estimates that organizations are losing twice as much revenue from slowdowns as they compared to availability. While slowdowns cost the business about 20% as much per hour as a full outage, organizations experience partial slowdowns 10x as often.

Adding 3rd-party providers inevitably increases this slowdown risk - what happens when any given external `<script>` tag is slow to load or times out?

- If possible, the included script must have an `async` attribute added. No script injection allowed. If the 3rd-party provider doesn’t provide a non-injectable version, it’s usually simple to make your own. For example, [here’s a non-script-injected Google Analytics tag](#).
- Set up aggressive timeouts and error handling if the 3rd-party provider goes down or suffers a partial outage. Simulate downtime by adding an incorrect URL to the “src” attribute - what happens? Does your page fail to load? Partially? Be defensive.
- Evaluate your 3rd-party providers. Google, for instance, is going to have a better uptime record than SomeNewStartupCo.
- Quantify your downtime risk. Say you expect your 3rd-party provider to have an uptime of 99%, and a “full uptime” (as in, full speed without slowdowns) 95% of the time (these numbers are typical). Figure out what it would cost your site during these downtimes. Is the projected benefit of the new integration greater than this cost?

Longer Load Times are an Expense

The correlation between webpage load times and conversions is one of the most well-studied and well-supported in web performance literature. Let's just get the quick hits:

| Business | Conversion Rate Increase per Second of Load Time |
|--------------------|--|
| Staples | 10% |
| AutoAnything | 2.6% |
| GlassesDirect | 7% |
| Etam | 35% |
| Walmart | 2% |
| Intuit | 2-3% |
| Shopzilla | 2% |
| Mozilla | ~7.5% |
| Bing | 2% |
| The Aberdeen Group | 7% |

This number will be different for every organization, but there's no doubt that the correlation exists. If you had to ask me for a rule of thumb, you could say that **for large (Fortune 500) companies, 1 second in load time equals a 2% change in conversion rate. For medium to small size organizations, 1 second equals a 7-10% change in conversions.**

Let's do a back of the napkin example.

ACME makes \$1 million/year and their site loads in 6 seconds on the average customer connection. Their conversion rate is 2%. Now let's say ACME can aggressively reduce that to 1 second. Perhaps they were able to ditch all of their 3rd-party Javascript and got smart about compressing their assets and reduced their page weight significantly. Based on the studies above, we would expect *at least a 10% improvement in conversions* for that 5 seconds of load time, and more likely we'd see a **70-100% increase in their conversion rate** - it's now 4% instead of 2%. You just made their \$1 million business into a \$2 million business. And heck, let's do the pessimistic 10% improvement scenario - conversion at 2.2% instead of 2%. That's still a \$100,000/year change. You just paid your salary.

Longer Load Times Turn Away Users

Conversion isn't the only thing that suffers when performance takes a dive. Not all business models are focused on "conversion" - ad-focused businesses need engagement and pageviews. Turns out that load times have a *huge* impact on these metrics as well.

| Business | Pageviews improvement per second of load time |
|--------------------|---|
| GQ | 11% |
| Etam | 40% |
| Yahoo | 22% |
| Shopzilla | 5% |
| The Aberdeen Group | 11% |

These are numbers that should make any marketing department drool. A 10% increase in traffic is not easy to come by at scale - but that's exactly what you'll get for reducing load times by just 1 second. Consider that most websites take 5-10 seconds to load when they could take just 1 second and you can see the massive opportunity available.

Mobile and Global Users are Hardest-Hit

"Mobile users" and "the international market" are buzzwords that marketers *love*. Ahh, but what would they do if they knew that the performance-sucking deadweight they love so much affected those users *the most*?

YouTube saw massive increases in traffic from South America, Asia and Africa after reducing page weight by 90% from 1.2MB to 100KB. Think about that - if YouTube found that a 1.2MB page is "mostly unusable" in those areas of the world, how do you think the average 2MB webpage performs? Yikes.

Etsy added just 160kb of images to a mobile site and saw a 12% increase in their bounce rate. Bandwidth on mobile is extremely scarce. Consider that the average bandwidth on a mobile connection in the United States is just 4 megabits/second. If you want your mobile website to load in 1 second or less, your page weight budget is vanishingly small. Subtract 300 milliseconds from your budget for latency and you're left with just 300kb, *absolute maximum*, and likely you'll need to use much less than that.

Akamai pegs the average US connection at 12 megabits/second (or just 1.5 megabytes/second). **Page weight targets for mobile users will need to be about 3x smaller than those for home broadband users because of this huge difference in**

available bandwidth.

Bandwidth is an Expense

This one only makes sense for the bigger companies, but the impact can be huge and the amount of effort required can be so low to fix it.

Netflix reduced their bandwidth bill by 43% just by turning on GZip. Whoops. I bet they wish they did that earlier.

Bandwidth ain't free. Calculate the expense of a served asset by multiplying its size by your traffic and your bandwidth costs. What you find out might surprise you.

Fast Sites are Cheap Sites

Thanks to Little's Law, we know that fast sites are easier to scale than slow sites because they require fewer resources. This makes intuitive sense - a site that takes 100 milliseconds to render on the server can serve (at maximum) about 10x as many requests per second as a site that takes 1000 milliseconds to render.

Ruby applications are usually memory-constrained - you *could* be running more instances of your app per server if only you increased its available RAM. **Memory-sipping Ruby apps are cheap Ruby apps.** Memory savings can translate into more application instances per server and, therefore, better server costs.

How To Create a Performance Culture

- **Quantify performance costs in dollars, not seconds.** Only other programmers understand and empathize with “this could be X% faster!”. Everyone, however, speaks the language of dollars and cents. If it feels weird to estimate costs based on research carried about by other companies, welcome to the business world.
- **Set a front-end load time budget:** If you have an existing project, fire up Webpagetest.org and take the average of 5 runs. `window.load` is not a great metric for this, but neither is `DOMContentLoaded` or Webpagetest's “SpeedIndex”. It's probably best to just set a budget for several metrics - I'd say `DOMContentLoaded` and `window.load` (in Webpagetest, “Document Complete” and “Fully Loaded”), and possibly “start render time” are fine places to start. Agree on this number as a team, record it somewhere for all to see. Agree that if your site exceeds these numbers, you'll file a bug. Increasing the budget should at least require the team's approval.

- **Set MART and/or M95RT:** Set a maximum average response time and/or a maximum 95th percentile response time for your server responses. Load times will follow a power law distribution rather than a simple normal distribution, so it's important to capture what's going on in the "long tail" as well as what's happening to the average case. Like your load-time budget, agree on this number as a team and for all to see. Quantify the costs of exceeding this budget - increased server requirements, increased user load times.
- **Set a page weight budget:** This one will be based on your load time budget and your user's average bandwidth. Your page weight cannot exceed / , and in fact should probably be set at about 50% of that number, because network utilization is never 100% while a web page loads. . For your calculations here, [Akamai estimates the following](#): worldwide average bandwidth is 0.625 megabytes/sec. US average bandwidth is 1.5 megabytes/second. Worldwide *mobile* average bandwidth is 0.1875 megabytes/second. US mobile average bandwidth is 0.5 megabytes/second.
- **Quantify integration costs:** Using the studies above, quantify the *dollar value* or *traffic value* (in visits/month) of a second of load time to your organization. Use this number when evaluating 3rd party integrations.
- **Add automated performance and page weight tests:** Add automated performance tests to your CI. Use my lesson to see how.

Checklist for Your App

- Use the process in this lesson to quantify the cost of an additional second of browser load time. Post this number where your team can see it. Discuss the process of how you arrived at this number with your team and whoever makes the business decisions.
- Set a front-end load time budget, and agree on a method of measurement. No, you won't be able to perfectly replicate an end-user experience - that's OK. Agree that load times exceeding this budget is a bug.
- Set a maximum acceptable response time and maximum acceptable 95th percentile time.
- Set a page weight budget based on your audience's bandwidth and the other budgets you've set.

¹:The exception is when a programmer gets to try out some new technology. Why build a simple Tic Tac Toe site when you can build it on Node with a MongoDB database and Websockets/Angular 2 front-end?!? Maciej's article calls this the Call of Duty web.

Performance Testing

Performance gets a lot of lip service. Designers and programmers constantly brag about how "lightweight" and "minimalistic" their code or designs are. And yet, the web is suffering from a bloat crisis. This [chickenshit minimalism](#) has saddled us with 2MB webpages, deep and complex DOM trees, and [the problem only appears to be getting worse](#).

Developers tend to work well with tight feedback loops. The more immediate the better: our test suites run on file changes in our apps, we run continuous integration servers to make sure no one broke the build. But why is performance testing (and its cousin, load testing) get short shrift? I spent some time searching for tools that could be considered "automated full-stack performance testing" for Ruby webapps, and came up with *almost* nothing. With great tools like [brakeman](#) and [rubocop](#) available for automated security and style checking, respectively, why does it seem like there isn't a comparable solution for performance or scale?

Why test performance?

Perhaps performance just isn't important enough to test? Well, as covered in my previous lesson, we know that isn't true. Slow websites can decrease conversion rates and drive traffic to competitor's sites as users get gradually fed up with bloated and slow interfaces.

As mentioned in that lesson, there is massive value in a team MART/maximum average response time. **Performance can be a feature: a fast and snappy user experience is just another user story, which can be elaborated and acceptance tested just like any other.** Setting a maximum average response time or similar performance "goals" helps get a team into a bug-finding-and-fixing mindset when it comes to performance issues. If there's no verifiable standard for performance, performance just can't "find a way in" to our usual developer workflows, and gets pushed aside to make deadlines and stay on track on burndown charts. Automated tests help provide a concrete, black-and-white standard for the performance "story" or feature.

When should I write a performance test?

The bare minimum performance test will start where your users start - your homepage. If, for some reason, your organization has users come in through a different page (check your analytics, don't guess), start your performance tests there.

A barebones full-stack performance test would simply make a GET request to this page and record two or three numbers:

- Server response time. In a test environment, where the server and client are on the same machine, this is the same thing as time-to-first-byte.
- User page load timings. `DOMContentLoaded` and `load` are probably the most important events, although these aren't always perfect analogues to "when the page becomes usable", which is the important thing.

An acceptance (some might call this an integration) test is a great place to start with a Ruby application. For any given action, about 50% of the code executed is going to be the same - each request passes through the same web server, the same Rack middleware stack, the same application configuration, the same database setup. By writing just one integration test, you've covered about 50% of the code in your application. That's great - eventually you'll probably want to keep writing these tests until you reach about 80% coverage, but one page is a great start.

Other than full-stack acceptance tests, what other kinds of performance tests should we seek to write? Every once in a while, you may have a "hot loop" in your application, or you may implement an algorithm which you know will be executed 1000 times or more in quick succession. Here are some examples of what I'm talking about:

- Once I implemented an algorithm for calculating Customer Lifetime Value for a SaaS application. The algorithm would take 10,000 customer objects or more and reduce them into a single average "lifetime value" - the algorithm that calculated this, if poorly implemented, could take several seconds to execute. I wrote a performance test to ensure that my implementation worked in linear time and was fast for the number of customers I expected would be input into the algorithm.
- Background jobs frequently perform operations against hundreds or thousands of database rows. For example, a job might update all of your customer's account balances. These jobs are great candidates for a quick performance test.
- If, when profiling, you find some "hot" code in your application that's executed many times during a request, be sure to write performance tests before fixing any performance-related bugs in that code. Without tests, regressions can happen at any time.

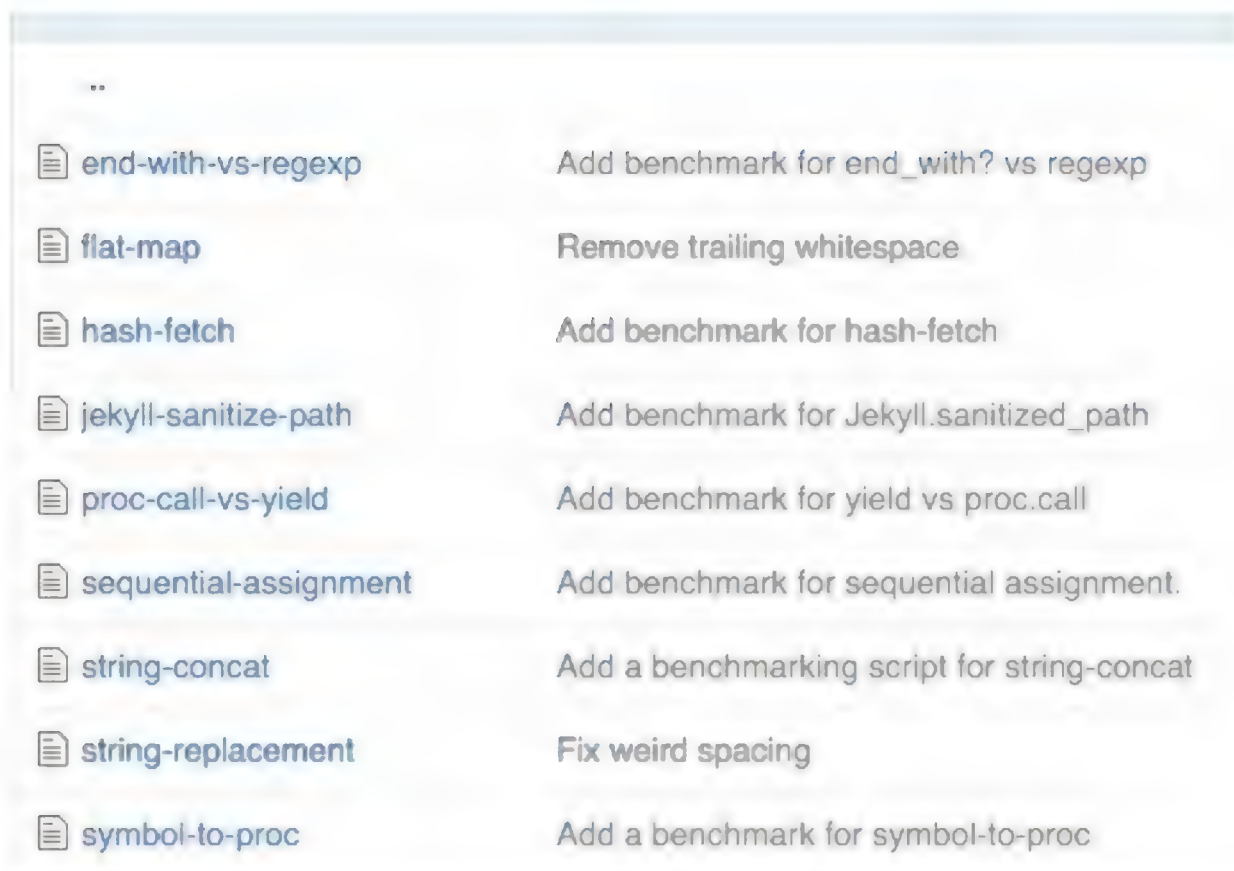
Baby's First Performance Test - the Benchmarks Folder










A common pattern for getting started with performance testing in Ruby applications is a `benchmarks` folder. Inside of the folder is a series of benchmark files, which can be run individually or as a suite. This pattern is most common in libraries, but could easily be adopted to work with a full application.

Benchmark folders are not "tests" per se - they don't contain assertions. Instead, benchmark folders are tools for developers to measure the impact of changes in the code. Run the benchmark suite against the master branch, then compare that against a benchmark run on the feature branch. By working off a shared set of benchmarks, everyone can agree on how to measure and optimize the performance of the app or library.

As an example, let's take a look at [the benchmarks folder for jekyll](#) - the popular blogging framework.

We see a number of benchmarks listed:



| | |
|--|---|
| .. | |
|  <code>end-with-vs-regexp</code> | Add benchmark for <code>end_with?</code> vs <code>regexp</code> |
|  <code>flat-map</code> | Remove trailing whitespace |
|  <code>hash-fetch</code> | Add benchmark for <code>hash-fetch</code> |
|  <code>jekyll-sanitize-path</code> | Add benchmark for <code>Jekyll.sanitized_path</code> |
|  <code>proc-call-vs-yield</code> | Add benchmark for <code>yield</code> vs <code>proc.call</code> |
|  <code>sequential-assignment</code> | Add benchmark for sequential assignment. |
|  <code>string-concat</code> | Add a benchmarking script for <code>string-concat</code> |
|  <code>string-replacement</code> | Fix weird spacing |
|  <code>symbol-to-proc</code> | Add a benchmark for <code>symbol-to-proc</code> |

A lot of these benchmarks are actually comparisons of different ways to write the same Ruby code. Here's `string-concat` :

```
require 'benchmark/ips'

url = "http://jekyllrb.com"

Benchmark.ips do |x|
  x.report('+=') { url += '/' }
  x.report('<<') { url << '/' }
end
```

Jekyll's maintainers add to this folder whenever they make a change in their codebase for performance reasons - for example, [here's one where they swapped regex matching with the `end_with?` method](#). In this way, the Jekyll project uses their `benchmarks` folder to compare and discuss performance changes on a micro-level.

Moving up the stack a little, they also sometimes add benchmarks to compare implementations of Jekyll features. For example, [this benchmark covers Jekyll's implementation of their `sanitized_path` method](#). In the future, if changes are made to the `sanitized_path` method, the benchmarks can be compared across branches.

These benchmarks can live in your test folder, though, too - [this is how `dalli`, the popular memcache client does it](#).

When you're ready to step up from the usual `benchmarks` folder, `Minitest::Benchmark` can be a useful tool, especially when double-checking the performance of algorithms or similarly "hot" code that operates over large collections. `Minitest::Benchmark` evaluates the performance of a method against increasingly large inputs - for example, you could benchmark a binary search against an Array of 10 elements, then 100, then 1,000, then 10,000, and so on. Minitest then uses a statistical regression test to see if your method performed the way you expected - in the case of binary search, logarithmically.

Performance acceptance tests

The next step for a web application would be adding some full-stack, "integration" style performance tests. There are a lot of ways to do this - unfortunately, I have to say, there is no "just add this to your Gemfile and add ten lines of config!" solution. We'll have to hack together what we need ourselves *or* use an external vendor service (covered in the next section).

An important note with performance acceptance tests - when setting the pass/fail threshold for a test like this, there's always going to be some grey area. Performance tests can never truly be deterministic, and will always depend on network conditions or CPU background load. For these reasons, I recommend the following with performance acceptance tests:

- If running in a testing environment (non-production), you can still compare results relative to each other. They'll almost certainly not compare accurately to the production environment, but that's OK - set a pass/fail standard based on the performance characteristics of your CI/test setup.
- Run the performance acceptance tests *seperately* from your unit and other acceptance/integration tests. You may want to allow them to fail or to automatically retry them in case of failure.
- You may decide that a hard pass/fail (that breaks the build) is not appropriate. In that case, just tracking the change in these performance test results over time is still valuable and could help you to track regressions down to certain commits.

Let's walk through a "poor man's performance test" using `wrk` (available on homebrew). Here's the command we'll use:

```
wrk -c <concurrent user count> -t <cpu core count> -d 10 -- latency http://our-server/
```

Remember, for these tests to work, you're going to need as production-like an environment as possible. Here's an example result:

```
nodachi:todomvc-turbolinks nateberkopec$ wrk -c 100 -t 4 -d 10 --latency http://localhost:3000
Running 10s test @ http://localhost:3000
 4 threads and 100 connections
Thread Stats   Avg      Stdev     Max    +/-  Stdev
Latency       57.40ms   10.15ms  123.95ms  76.11%
Req/Sec      139.42     56.24   242.00    54.00%
Latency Distribution
 50%    55.41ms
 75%    62.03ms
 90%    70.52ms
 99%    90.06ms
2788 requests in 10.04s, 9.22MB read
Requests/sec:   277.61
Transfer/sec:    0.92MB
```

When benchmarking like this, it's important to know that the database matters. Data should be either a *copy of production*, if at all possible, or *similar in size to production*. This could require either the creation of a sizeable/robust `seed.rb` file in Rails or some sort of anonymizer/masker script to clean your production database. It's important that calls that interact with the database return the *same amount of rows they would in production*, otherwise you're going to miss slowdowns from queries that only return a handful of rows in development but several hundred in production.

Once you have the output of a benchmark like this, you can turn it into a pass/fail test with some Unix magic in a script, like so:

```
#!/bin/bash
MINIMUM_REQ_SEC = 100
wrk -c 100 -t 4 -d 10 --latency http://localhost:3000 > /tmp/perf_test.txt
(( grep "Requests/sec:" /tmp/perf_test.txt | awk '$2 >= $MINIMUM_REQ_SEC' )) || {
  echo "Performance test failed"; exit 1; }
```

A simple script like this could be added to any CI server's flow.

3rd-Party Services: Blazemeter and Loader.io

While you can certainly set up your own performance tests on your own CI servers, there are several third-parties that offer this as a service now. Unfortunately, the tools available for "building your own" performance tests are pretty limited - it doesn't get much more complex than the `wrk` script example above. However, some vendors offer some quite extensive performance tests. I'll use my TodoMVC implementation as an example app to show you the differences between two of the major vendors - Blazemeter and Loader.io.

Most of these services require a URL to be tested against - my advice is to use your staging server and put it behind HTTP Basic Auth.

Blazemeter

[Blazemeter](#) is the "big boy" solution of this space. It's essentially a fancy front-end for [JMeter](#), an Apache Foundation project for load testing. It's available as a Heroku add-on.

Blazemeter tests against a public URL - so you'll need a staging site or something like that, since you don't want to load down your production setup. [Heroku's review apps](#) feature may also be appropriate for this. Just enter in a URL you want to test, and BlazeMeter will pound it with concurrent users. [It looks like Blazemeter now supports testing through a firewall as well, in case you'd like to test apps not available at public URLs.](#)

Blazemeter's default test configuration is a long, slow ramp over 15 minutes - starting from 1 user, it ramps up until it simulates up to 1000s of users hitting your site at the same time.

The featureset is extensive and the reports are extremely detailed - it's clearly The Enterprise Solution in this space. It supports extensive scripting through JMeter as well - you could load test an entire checkout flow, for example. It looks like you can even use Selenium tests to drive the load tests as well. Of all the providers listed here, Blazemeter clearly had the most robust scripting features - or indeed, most robust features in total. All of those features don't come cheap though - basic plans start at \$150/month.

Loader.io

Loader.io is a simple load-testing service by the people at SendGrid.

Also available as a Heroku add-on, I found Loader.io's interface quite simple and easy to use. Even their free plan had a useful amount of features and could generate a decent amount of load (250 concurrent users over 1 minute).

However, I found Loader.io's overall featureset pretty lacking. While it was simple to set up and understand, it seemed like every feature was offered by Blazemeter too. And Blazemeter is only \$50 more per month.

Checklist for Your App

- Run a benchmark on your site locally using `wrk`.
- Set up a performance monitor/tester for your application. Ensure it either runs continuously or after every deploy. Self-rolled or 3rd-party is fine.

Lab: Performance Testing

Exercise 1

Let's use Apache Bench to benchmark the homepage of a local instance of Rubygems.org. If you haven't set up Rubygems.org yet, see `RUBYGEMS_SETUP.md` in the root of this project.

Start up Rubygems.org and install Apache Bench. It generally goes under the "ab" name in package managers - "brew install ab" to give it a shot.

Apache Bench, like most load-testing tools, allows you specify a number of requests to perform concurrently, supports HTTP basic auth (good for staging servers), SSL, proxies, and setting custom headers.

When using Apache Bench, I usually use a concurrency setting somewhat equal to what I'd expect in production (~10 users on my site at one time, for example) and run the test for about a minute:

```
ab -c 10 -t 60 http://127.0.0.1:3000/
```

After running the benchmark, what conclusions can you draw from the results?

Experiment with running Rubygems.org with different application servers, such as Puma and thin (Rubygems.org will use Webrick by default). Does changing the application server make any meaningful impact on results?

Solution

- Notice the noise in the results, even though this benchmark renders the *exact* same page every time. I received a mean time of 625ms, with a standard deviation of 81 milliseconds - that's a decent size deviation for so simple a task!
- Puma makes a considerable impact on the result here - when using `bundle exec puma config.ru -w 4 -t 8:8` to run Puma with 4 workers at 8 threads each, I was able to process twice as many requests in the 60 second benchmark.
- Thin also performed better than Webrick, though not by much - just 10% more requests/sec on my machine.

Profiling with Ruby-Prof, Stackprof and GPerfTools

Where does all the time go?

It's 9 a.m. You sit down at your desk, warm coffee in hand, and pop open New Relic or Skylight to take a look at your performance metrics. You do a spit-take - 500 millisecond average response times?! The 95th percentile is over 10 seconds?!

What do you do now?

Both Skylight and New Relic have some built-in **profilers**, which can start to shed some light on where all the time goes during each of your requests. While these profilers are useful (since they work in the real production environment, with real users and real requests), they will always be limited by the fact that they're not interactive. They only live in the past. If we want to make changes, then see what changes in the profile, we have to wait a day or so for new production users to try our new code. That's not helpful - and what if we make the *wrong* change and make it worse?

I discussed in a previous lesson, the difference between profiling and benchmarking:

Profiling is great - it's one of the most important tools we have for answering "Well, what next?" once we determine that a benchmark result or performance metric is unsatisfactory. Profiling points us to the bottlenecks and answers "Where does all the time go?"

However, it should be noted that profilers themselves nearly always impose a performance cost on an application. This makes sense - in addition to doing the work it had to do before, adding a profiler means the program has to do the work *and* keep extremely meticulous records of *how it did it*. Remember how annoyed you were in grade school when your teachers asked you to "show your work"? Same thing.

Profiling and Benchmarking - A Workflow

When diagnosing performance issues, my workflow generally looks like this:

1. **Explore metrics for problems.** Are any metrics on production in violation of my team's agreed-upon performance standards? Anything out of the ordinary?

2. **Establish a reasonable benchmark.** Once I observe the performance issue “in the wild”, the next step is creating a benchmark on my local machine that approximates the same issue on production. For example, if I notice that our homepage is rendering slowly, I might create a benchmark suite that creates a server locally, tests that server with production-like load, and outputs the average response time in milliseconds. It’s important to note that the benchmark time here will never exactly match the time in production - your hardware is different, of course. But as long as we’re in the same neighborhood, that’s fine (500ms in production vs 1000ms locally, for example. Generally a factor of 5 is acceptable).
3. **Profile, and figure out where the time goes.** Why is it slow? Running a profiler on our benchmark code will give us an idea of *where the time goes*. We’re spending 50% of our time in the `Integer#times` method? We’re spending 75% of our time making *database calls*?
4. **Experiment with and benchmark alternatives.** Now that we know what our code is doing, we can try speeding up the parts that are slow. Profilers are never 100% accurate - if the profiler says we spend 50% of our time in X method, removing X method may *not* speed up our benchmark results by 50%. More on why in a bit. But after making a change based on our profiler results, we should always re-benchmark and re-profile to make sure our hypothesis was correct.

This is the scientific method, dressed up a bit.

1. **Make observations.**
2. **Formulate hypotheses.**
3. **Develop testable predictions.**
4. **Gather data to test predictions.**

When Profilers Lie - Choosing a Profiler Mode

Nearly every profiler will ask you to choose a *mode*. One of the biggest problems with profiling is agreeing upon a unit of *time*. Profilers are fundamentally about asking which lines of code take up what proportion of execution time, but the exact way we want to measure that is not always clear. There isn’t really a “right answer” for what profiler mode you’re going to want to use in any given situation. Not all profilers support all of these different approaches.

Let's take a look at common profiler mode settings and the advantages and disadvantages of each.

CPU - clock counter

Consider the following code:

```
require 'ruby-prof'

def sleeper
  sleep(4)
end

# We'll get to RubyProf in a minute, just know that I'm profiling
# whatever is in the given block.
RubyProf.measure_mode = RubyProf::CPU_TIME
result = RubyProf.profile { sleeper }
printer = RubyProf::FlatPrinter.new(result)
printer.print(STDOUT)
```

The output is going to be a little weird:

```
Measure Mode: cpu_time
Thread ID: 70350977366560
Fiber ID: 70350977841000
Total: 0.000092
Sort by: self_time
```

| %self | total | self | wait | child | calls | name |
|-------|-------|-------|-------|-------|-------|--------------------|
| 50.00 | 0.000 | 0.000 | 0.000 | 0.000 | 1 | Kernel#sleep |
| 39.13 | 0.000 | 0.000 | 0.000 | 0.000 | 1 | Global#[No method] |
| 10.87 | 0.000 | 0.000 | 0.000 | 0.000 | 1 | Object#sleeper |

Take a look at the “total” column - ordinarily this is a measure of how much time we spent in a method. And it's... zero? But I know this takes at least 4 seconds to execute! What's going on?

`sleep` suspends the current thread - which means while we're sleeping, we're not using clock cycles in the current thread. The profiler has nothing to measure in CPU mode! CPU time *counts the number of clock cycles* to measure time, rather than counting seconds. There are no seconds or nanoseconds - just numbers of CPU instructions. Consider instead what happens when we give the thread some busywork for 4 seconds instead:

```

require 'ruby-prof'

# https://github.com/ruby-prof/ruby-prof/pull/137
def sleeper
  t = Time.now.to_f
  while Time.now.to_f < t + 4.0
    # busy loop
  end
end

RubyProf.measure_mode = RubyProf::CPU_TIME
result = RubyProf.profile { sleeper }
printer = RubyProf::FlatPrinter.new(result)
printer.print(STDOUT)

```

And our output is:

```

Measure Mode: cpu_time
Thread ID: 70355813210680
Fiber ID: 70355809515860
Total: 3.954522
Sort by: self_time

```

| %self | total | self | wait | child | calls | name |
|-------|-------|-------|-------|-------|--------|--------------------|
| 20.30 | 1.772 | 0.803 | 0.000 | 0.969 | 657015 | <Class::Time>#now |
| 16.38 | 0.969 | 0.648 | 0.000 | 0.321 | 657015 | Time#initialize |
| 16.18 | 1.579 | 0.640 | 0.000 | 0.939 | 657015 | Time#to_f |
| 15.70 | 0.939 | 0.621 | 0.000 | 0.318 | 657015 | Numeric#quo |
| 15.28 | 3.954 | 0.604 | 0.000 | 3.350 | 1 | Object#sleeper |
| 8.11 | 0.321 | 0.321 | 0.000 | 0.000 | 657014 | Fixnum#+ |
| 8.04 | 0.318 | 0.318 | 0.000 | 0.000 | 657015 | Fixnum#fdiv |
| 0.00 | 3.955 | 0.000 | 0.000 | 3.954 | 1 | Global#[No method] |

Now we can see that RubyProf correctly thinks that we're spending *approximately* 4 seconds inside of the #sleeper method, which is what we would expect.

Unfortunately, modern CPUs (especially those in power-constricted environments like laptops) do a lot of what's called *stepping*. This means they change their clock speed based on the load they're under - when under high load, they speed up, when under low load, they wind down. Higher clock speeds equal higher power consumption. However, this can throw off CPU times - since CPU time is basically "Amount of clock cycles" / "CPU frequency", if the CPU frequency changes during your benchmark, you're gonna have a bad time. Disabling CPU stepping isn't usually a good idea though, and could cause your poor little laptop to overheat or at least sound like a helicopter as its poor little CPU fan attempts liftoff.

Most CPU time measurements are *system-wide*, meaning that we are estimating CPU time based on the [time stamp counter](#). Thus, background work on your operating system *will* affect the CPU time - if your system is under load (you're watching cat videos while your profiler runs, for example), the times in your profile will increase.

Use CPU time when you're interested in seeing the profile sans I/O. If profiling an operation that waits on a lot of I/O, such as a network request or an external database, CPU time can eliminate that time from your profile. However, you can also do this by excluding certain Ruby modules/methods from your profiling results (if the profiler allows it).

Don't use CPU time when you need really accurate results. Speed-stepping will almost certainly screw with your results. Run the profile a few times, at the least, and take the smallest result ([why not the average? here's a good explanation.](#)) In addition, you can disable speed stepping on your laptop for improved accuracy, or just profile your code while pegging your CPU at max (e.g. don't run a single request against your Rails app and profile it, run 10,000 requests).

Finally, there is a bug in Mac OS X's kernel that affects many CPU timing profilers - gperftools and stackprof, two profilers we'll cover later, are affected. As far as I can tell, however, ruby-prof uses a different mechanism and should be as accurate as is possible with CPU timing.

Wall time

"Wall" time refers to using the clock "on the wall", as if we had a microsecond-level clock on the wall of our office and could use it to measure the amount of time elapsed from one method invocation to another. This is real world, "stopwatch" time.

If we profile `sleep` with wall time, we'll get actually accurate output:

```
Measure Mode: wall_time
Thread ID: 70311995300360
Fiber ID: 70311999772520
Total: 4.003320
Sort by: self_time
```

| %self | total | self | wait | child | calls | name |
|--------|-------|-------|-------|-------|-------|--------------------|
| 100.00 | 4.003 | 4.003 | 0.000 | 0.000 | 1 | Kernel#sleep |
| 0.00 | 4.003 | 0.000 | 0.000 | 4.003 | 1 | Global#[No method] |
| 0.00 | 4.003 | 0.000 | 0.000 | 4.003 | 1 | Object#sleeper |

Internally, most wall time profilers use something like `gettimeofday`.

However, just like CPU time, wall time can be greatly affected by external factors:

- **Other processes (resource contention).** If the code you're profiling depends on accessing the disk, and you happen to be torrenting some Game of Thrones episodes at the same time, it's going to look like your program spends a lot of time waiting on I/O.
- **Network or I/O conditions.** Let's say you're profiling some code for a cache store, and that cache store uses Redis. When using wall time, your final results will be affected by whatever performance characteristics your Redis database has (it'll show up as a lot of time spent in `IO.select` or similar). This isn't really relevant most of the time, and usually obscures the Ruby code that takes long amounts of time to execute.

Despite its flaws, wall time is usually the mode you'll want to use. It's generally the most accurate tracing method for Ruby code that doesn't do a lot of I/O, and as long as you're not watching cat videos at the same time, it's fairly accurate.

Don't use wall time when I/O is involved and highly variable. For example, profiling code that accesses the network.

Process time

`ruby-prof`'s README does a good job explaining what process time measurement is supposed to do:

Process time measures the time used by a process between any two moments. It is unaffected by other processes concurrently running on the system.

Ruby-prof uses the `clock` function for this. You may be thinking this sounds great - it's like CPU time measurement, but better! Process time should be unaffected by other processes on the system, eliminating one of CPU time's major drawbacks, right?

Process time doesn't include time spent in *child processes*, so anything that uses `fork` or `spawn` is out.

However, aside from that caveat, **process time, if available, is usually a better choice over CPU time.** If you have code that spawns subprocesses, you may need to stick with CPU time (or wall time).

Process time, like CPU time, can be rendered inaccurate by CPU stepping.

Note that in some profilers, process time is simply called "CPU time". Be sure to note in your profiler's documentation how its CPU time function works.

Tracing

Some Ruby profilers are *tracers*, which means that they try to be as accurate as possible by trying to measure *every* method invocation and how long they take, and then *aggregate* that data across your entire profiling run. On the upside, this approach is extremely accurate. Unfortunately, it also usually comes with high overhead, making it inappropriate for production usage.

As far as I know, the only Ruby profiler that takes this approach is RubyProf. [You can see where it hooks directly into Ruby's execution here.](#)

If you've heard of DTrace, you should know that [MRI Ruby supports DTrace](#) as of version 2.0.

Sampling

All of the approaches we've discussed so far involve *aggregation*. As I said, this has the advantage of being extremely accurate, but the disadvantage of having high overhead. This means aggregating profilers are inappropriate for production use. However, there's another approach: sampling.

Just halt [your program] several times, and each time look at the call stack. If there is some code that is wasting some percentage of the time, 20% or 50% or whatever, that is the probability that you will catch it in the act on each sample. So that is roughly the percentage of samples on which you will see it. There is no educated guesswork required. If you do have a guess as to what the problem is, this will prove or disprove it.

Yes, by just randomly "CTRL-C"-ing during program execution and looking at the call stack, congratulations, you're sampling! The problem with this method, of course, is that you're taking an extremely small sample. Sampling profilers sample hundreds of times *per second*, giving us much higher resolution and accuracy than our "just halt it!" sampling method.

If you want to learn more than you ever wanted to know about how sampling profilers work, [check out this article](#).

So the “numerator” of any sampling profiler, instead of a unit of time, is actually “number of times this method/line-of-code appeared when we randomly sampled the stack.”

The *denominator*, however, can be either CPU time or wall time, with all the caveats of the above.

In general, use sampling profilers for production, and use aggregating/tracing profilers for development.

ruby-prof

As mentioned above, [ruby-prof works by hooking into MRI directly](#) - every time you call a method or something happens in the Ruby VM, ruby-prof gets called and measures how long it took the CPU to do that thing. This process is pretty intense - when running ruby-prof, expect your program to run 2-3x slower than it would otherwise. This makes it way too difficult to deploy ruby-prof sanely in production. Instead, you’ll want to use one of the sampling profilers I talk about later in this lesson.

`ruby-prof` is great for profiling in development. What else is it good at? In general, I tend to reach for a profiler like ruby-prof when I’m investigating code outside of the typical Rack request/response scenario. If I just want to profile a single request, typically I’ll just reach for rack-mini-profiler, which is far easier to use than any of the profilers in this lesson.

But what if you need to profile some Ruby code outside of a Rack scenario - say a test suite? Or perhaps a library? Or maybe rack-mini-profiler isn’t providing the detail you need?

ruby-prof helps us with these problems. Your benchmark says your code is slow - now what?

Quick and dirty profiling

Let’s compare two different implementations of binary search with ruby-prof and benchmark-ips. It doesn’t really matter how they’re implemented - we’ll just say it’s two methods, `bsearch1` and `bsearch2`. First, I’ll write a benchmark:

```
require 'benchmark/ips'

Benchmark.ips do |x|
  SORTED_ARRAY = Array.new(10_000) { rand(100_000) }.sort!
  array_size = SORTED_ARRAY.size
  # Typical mode, runs the block as many times as it can
  x.report("bsearch1") { bsearch1(SORTED_ARRAY, rand(array_size))}
  x.report("bsearch2") { bsearch2(SORTED_ARRAY, rand(array_size))}

  x.compare!
end
```

Note how I'm careful that the only code running in the loop is the bsearch method. We don't want to calculate the size of the loop every time, or worse, generate a new array every time. And here's the output:

```
Calculating -----
      bsearch1    54.917k i/100ms
      bsearch2    31.073k i/100ms
-----
      bsearch1    782.667k (± 2.7%) i/s -      3.954M
      bsearch2    374.304k (± 2.4%) i/s -      1.895M

Comparison:
      bsearch1:    782666.6 i/s
      bsearch2:    374303.7 i/s - 2.09x slower
```

We know bsearch1 is twice as fast as bsearch2. Awesome. But let's say we want to speed up bsearch2 - so let's profile it using ruby-prof to figure out why bsearch2 is so slow.

```
require 'ruby-prof'

SORTED_ARRAY = Array.new(10_000) { rand(100_000) }.sort!
array_size = SORTED_ARRAY.size

RubyProf.measure_mode = RubyProf::CPU_TIME

result = RubyProf.profile do
  1_000_000.times { bsearch2(SORTED_ARRAY, rand(array_size)) }
end

printer = RubyProf::FlatPrinter.new(result)
printer.print(STDOUT)
```


And here's the output:

| %self | total | self | wait | child | calls | name |
|-------|--------|--------|-------|--------|----------|--------------------|
| 17.22 | 19.117 | 12.777 | 0.000 | 6.340 | 13182869 | Fixnum#== |
| 8.54 | 6.340 | 6.340 | 0.000 | 0.000 | 13182869 | BasicObject#== |
| 5.73 | 71.918 | 4.252 | 0.000 | 67.666 | 14182869 | *Object#bsearch2 |
| 2.14 | 74.196 | 1.590 | 0.000 | 72.606 | 1 | Integer#times |
| 0.93 | 0.688 | 0.688 | 0.000 | 0.000 | 1000000 | Kernel#rand |
| 0.68 | 0.508 | 0.508 | 0.000 | 0.000 | 1000000 | Array#count |
| 0.00 | 74.196 | 0.000 | 0.000 | 74.196 | 1 | Global#[No method] |

Here's what the headers here mean:

- `%self` is the percentage of the total time elapsed spent in this method call
- `total` is the total time (in seconds, I think, but I'm honestly not sure) spent in this method *and its children* (other methods called by this method)
- `self` is the time spent in this method, excluding child method calls.
- `wait` is the time spent waiting here.
- `child` is the time spent in child method calls. `total - self = child`.
- `calls` is the number of times this was invoked.
- `name` is self explanatory.

It looks like whatever in this method is calling `==` is pretty hot. Also, it looks like `bsearch2` is implemented recursively - it appears to be calling itself. I'll start by trying to eliminate the method call to `==`. I do that, then retry my benchmark to see if I've improved anything - and whaddya know, `bsearch2` is now just 1.5x slower than `bsearch1` ! I could repeat this process until I was satisfied.

Generally, this is what profiler workflow looks like - you start with the biggest "`%self`" and keep optimizing your way downward.

Additional profiler modes

Aside from being able to run in CPU, process, and wall-clock modes as explained above, ruby-prof has some additional modes for measuring things like memory allocation. Unfortunately, these parts of ruby-prof are almost completely broken and unmaintained.

See my lesson on profiling memory usage for better alternatives.

Printer modes

ruby-prof features several different ways of printing output, but one of my favorites is the `CallStackPrinter`. It's extremely useful when dealing with Rack applications - to see it in action, check out my lesson on slimming down Rails.

stackprof

`stackprof` is a sampling profiler for Ruby 2.1+. This is the profiler used under the hood by `rack-mini-profiler`. Unlike `ruby-prof`, it samples rather than aggregates, making it usable in production environments.

I don't find `stackprof` useful in development, since `ruby-prof` usually works more accurately and has most of the features of `stackprof`. However, if you were looking at run your own profiles in production, this might be an interesting tool. Take a look at [stackprof-remote](#) if using `stackprof` to profile a Rails or Rack application - it's a tool that allows you to profile an app with `stackprof` while it's running.

`stackprof` supports wall and cpu timing modes. Note that `stackprof`'s method of CPU timing is bugged on Mac OS X (see above).

gperftools/perftools.rb

`perftools.rb` is a Ruby front-end for Google's [gperftools](#). `gperftools` is a sampling profiler, like `stackprof`, however, it's a "big boy" tool written and used by Google.

While, as far as I can tell, mostly similar to `stackprof`, `gperftools` has great graphical report modes. [Check out this sweet visualization of a Rails app](#).

`perftools.rb` runs in CPU mode by default. It is affected by the same CPU timing mode bugs on Mac OS X as is `stackprof`, so you may want to use wall mode instead.

Checklist for Your App

- Use a profiler like `ruby-prof` to diagnose your application's startup time. Where does most time go during your app's initialization process?
- Find an algorithm or other "hot" Ruby code in your app. If you can't, find two different implementations of a common algorithm (for example, binary search) online. Use a combination of benchmarks and profilers to determine why one implementation is faster than the other, and try writing your own optimized version.

Lab: Profiling

Exercise 1

Let's profile a benchmark suite.

We'll use Dalli's benchmark suite for this example. Dalli is a Ruby client for Memcache.

1. `git clone https://github.com/petergoldstein/dalli.git`
2. `git checkout 65a5a8c`
3. `gem install 'ruby-prof'`
4. Install and start a Memcache server.
5. `bundle install`
6. `ruby-prof test/benchmark_test.rb` will run ruby-prof around your test run.

Inspect the output. What does it tell you about Dalli? Use `ruby-prof --help` to investigate different options available to you, be sure to try to use concepts from the lesson.

Solution

- Dalli spends over 10% of its time waiting on IO - this makes sense, because it's fundamentally a library for talking to a database.
- The slowest method in Dalli is `Dalli::Ring#binary_search`. When using CPU profiling mode, you can see that this single method takes up almost 14% of the test run.
- We spend a lot of time in the `MonitorMixin` class - this is part of the standard library.

Profiling Ruby Memory Usage

In our previous lesson, we talked about profiling Ruby code. In addition to profiling execution time, we can also profile memory usage. A common barrier to scaling Ruby applications is bloated memory usage - and one of the first steps to fixing memory bloat is to understand where all of that memory is being used. This is what memory profiling is all about.

Unfortunately, the Ruby community is only *just* waking up to the possibilities afforded by memory profiling. Most of the tools are new and not well documented, and have limited feature sets. Memory profiling, unlike CPU profiling, is usually locked to the language VM. With CPU profiling, we can use Google's `perftools` profiler pretty much unmodified to profile Ruby. This isn't the case with memory - different language virtual machines manage memory in different ways, and memory profilers have to be aware of this.

This lesson is targeted at MRI Ruby, mostly because if you're on JRuby you're in much better shape. JRuby, as it runs on the Java Virtual Machine, can use any memory profiler that works on the JVM. These tools, unlike the tools for MRI Ruby, are much more mature.

There are five main tools we can use for memory profiling in MRI Ruby:

- **ObjectSpace and `objspace.so`** - introduced in Ruby 1.9 and improved and expanded since, `ObjectSpace` is a module in Ruby's `stdlib` that allows limited object and memory introspection.
- **`GC::Profiler`** - The `gc` module, also in Ruby's `stdlib`, has included a `Profiler` module since Ruby 1.9. It provides a lot of interesting statistics about garbage collection.
- **`gc_tracer`** - Written by Ruby core member Koichi Sasada, `gc_tracer` is an in-depth tracer for the new Ruby 2.1 garbage collector.
- **`derailed_benchmarks`** - Written by Heroku man [Richard Schneeman](#), this swiss-army knife provides some excellent memory-related profiles.
- **`memory_profiler`** - Product of Discourse's speed-demon [Sam Saffron](#), `memory_profiler` provides several in-depth memory profiling modes.

ObjectSpace and `objspace.so`

Since Ruby 1.9, Ruby has included a magical little module in a file called `ObjectSpace`.

```
irb(main):001:0> ObjectSpace.count_objects
=> {:TOTAL=>53802, :FREE=>31, :T_OBJECT=>3373, :T_CLASS=>888, :T_MODULE=>30, :T_FLOAT=>4, :T_STRING=>36497, :T_REGEXP=>164, :T_ARRAY=>9399, :T_HASH=>789, :T_STRUCT=>2, :T_BIGNUM=>2, :T_FILE=>7, :T_DATA=>1443, :T_MATCH=>85, :T_COMPLEX=>1, :T_NODE=>1050, :T_ICLASS=>37}
```

Neat, huh? Try running that in an `irb` session a few times and you'll even see the numbers grow! Most of these should be pretty self explanatory (`T_CLASS` , `T_OBJECT` , etc). The weird ones (`T_NODE` , `T_DATA`) are more for the interpreter's benefit than our own - `T_NODE` is counting the nodes of the abstract syntax tree of your program, for example. Just pay attention to the Ruby primitive types here. You can already see some applications for `ObjectSpace` - logging the `ObjectSpace` counts to an external service, for example, or to your development console as you click through your site. You can do this without any performance overhead worries - these statistics are already kept whether or not you are using them.

Did you know you turn Ruby's garbage collector on and off? It's really simple:

```
GC.disable #=> true, GC is now disabled
GC.enable  #=> true, GC is now enabled.
GC.start   #=> garbage collect RIGHT NOW
```

You can combine `ObjectSpace.count_objects` with `GC.start` to test your own theories about how the garbage collector works on a micro scale. For example, how many strings will this allocate?

```
100.times do
  'hello' + ' ' + 'world'
end
```

First, we'll write a method to determine what objects are allocated in any given block:

```
def allocate_count
  GC.disable
  before = ObjectSpace.count_objects
  yield
  after = ObjectSpace.count_objects
  after.each { |k,v| after[k] = v - before[k] }
  after[:T_HASH] -= 1 # probe effect - we created the before hash.
  after[:FREE] += 1 # same
  GC.enable
  after.reject { |k,v| v == 0 }
end
```

We disable the garbage collector, count the objects, yield to a given block, recount all the objects, diff the counts, and re-enable garbage collection. We turn GC off, because if the garbage collector turns on while we're counting the objects, that would totally mess up our count. Let's check the result:

```
irb(main):042:0> allocate_count { 100.times { 'hello' + ' ' + 'world' } }
=> {
  :FREE => -500,
  :T_STRING => 500
}
```

Did you get the right answer (500)? If not, here's a hint: Ruby combines the strings one at a time - "hello" + " " becomes "hello ", and so on.

Using this `allocate_method`, we can "micro-benchmark" different idioms to see which ones use more memory than others. You can also learn a lot about how Ruby memory works this way too.

Oh - and check *this* out.

```
puts ObjectSpace.each_object.count #=> 42552
puts ObjectSpace.each_object(Numeric).count #=> 7
puts ObjectSpace.each_object(Complex).count #=> 1
ObjectSpace.each_object(Complex) { |c| puts c } #=> 0+1i
```

That's right - you can iterate through *every live object*. Let's see your language of choice do *that*!

There are lot of applications for `ObjectSpace.each_object`. `minitest` originally used it to discover test classes and methods (it doesn't anymore). You can use it in development to count and inspect objects created in your application - for example, open up a console

mid-request (with something like [pry](#), [webconsole](#), or [better_errors](#) and start counting and iterating through ActiveRecord objects.

For example, here's a way to print all active objects by class, giving you an idea of what modules are creating and retaining the most objects:

```
ObjectSpace.each_object.
  map(&:class).
  each_with_object(Hash.new(0)) { |e, h| h[e] += 1 }.
  sort_by { |k,v| v }
```

Try this example yourself and pay attention to the output - it will look familiar once we get to `memory_profiler`.

`require "objspace"` extends the `ObjectSpace` module with several awesome methods. It's a kind of "debugging" module, really - the documentation comes with this stern warning:

Generally, you *SHOULD NOT* use this library if you do not know about the MRI implementation. Mainly, this library is for (memory) profiler developers and MRI developers who need to know about MRI memory usage.

There's a good reason for this. `require "objspace"` will slow any production application to a crawl, thanks to all of the tracing it adds, so this is strictly for development use only. With that caveat, `ObjectSpace` has a lot of superpowers. You can read about all of them in [the official documentation](#), but I'm going to show you the ones that are really useful for any Ruby developer that's trying to understand how their app uses memory.

Here's an interesting one:

```
irb(main):057:0> ObjectSpace.count_objects_size
{
  :T_OBJECT => 198560,
  :T_CLASS => 614784,
  :T_MODULE => 66712,
  :T_FLOAT => 160,
  :T_STRING => 1578522,
  :T_REGEXP => 122875,
  :T_ARRAY => 630976,
  :T_HASH => 165672,
  :T_STRUCT => 160
  ...
}
```


`ObjectSpace.count_objects_size` shows you, in bytes, how much memory each type of object is using.

We can check the size of objects (in bytes) with `memsize_of` :

```
irb(main):062:0> ObjectSpace.memsize_of("The quick brown fox jumps over the lazy dog")
40
irb(main):063:0> ObjectSpace.memsize_of("The quick brown fox")
40
irb(main):064:0> ObjectSpace.memsize_of([])
40
irb(main):065:0> ObjectSpace.memsize_of(Array.new(10_000) { :a })
80040
```

Note how this demonstrates a bit of how CRuby uses memory that you may not have been aware of - objects are pretty much always at least 40 bytes.

There's also `memsize_of_all` to get the total memory size of a certain class of objects - this is slightly more useful than `ObjectSpace.count_objects_size` because it actually uses the classes in your application, rather than the internal data types of MRI:

```
irb(main):066:0> ObjectSpace.memsize_of_all(String)
600682
```

Use ObjectSpace for: Playing around and enhancing your knowledge of how Ruby creates and garbage collects objects. Use `ObjectSpace.each_object` to explore and introspect currently live objects in your application. If any of the tools I cover in this lesson don't exactly fit your needs, you can usually hack ObjectSpace into a mini-tool that gives you the output that you want.

GC::Profiler

`GC` , as mentioned above, is just a module in the stdlib for working with the garbage collector. `GC::Profiler` , in the words of the documentation, provides access to information on garbage collector runs, including time taken and object sizes.

`GC::Profiler` is included by default, so we don't need to `require` anything (unlike `objspace`).

Before we get to `GC::Profiler` , though, let's talk for a quick second about two more methods available on `GC` :

- `GC.count` returns an integer that's simply the number of times the GC has run since the process started. This includes major and minor GC runs. Since Ruby 2.1, MRI Ruby has implemented a *generational garbage collector*, which means that objects are flagged based on how many garbage collections they've survived. When an object survives a garbage collection, it's marked "old". Minor GCs, as opposed to "Major" GCs, only attempt to garbage collect "new" objects. The principle is that most objects die young - we don't need to check old objects as often as we check new ones. `GC.stat` is useful for checking if GC occurs during a request or during some work.
- `GC.stat` outputs a detailed hash with some details on garbage collection. Aside from the count, which I've already described, you'll see lots of details about the heap, free memory, and more. There's a lot in here, and most of it won't make any sense unless you know a lot about how Ruby's GC works. What I want to call your attention to is `minor_gc_count` and `major_gc_count`, which breaks out the total GC count into minor and major runs. In addition, the `old_objects` key can be useful for tracking down leaks - if `old_objects` is gradually increasing over time, it could be a memory leak.

Anyway, let's talk about `GC::Profiler` for a second. As usual, the performance cost here is high - unlike `GC.count` and `GC.stat`, which have zero overhead, `GC::Profiler` use in production is not advised.

Like most profilers, you can run `GC::Profiler` with `enable` and `disable` methods, like so:

```
GC::Profiler.enable

require 'set'
GC.start

GC::Profiler.report
GC::Profiler.disable
```

As you can see, I forced a GC run here with `GC.start`. If a GC doesn't run between `enable` and `disable`, `report` will return nil. Here's the output of the above:

```
GC 133 invokes.
Index   Invoke Time(sec)      Use Size(byte)      Total Size(byte)      Total
Object                                GC Time(ms)
  1             1.966             801240             6315840
157896      2.337000000000003349498
```

Note that the "invokes" number there is just `GC.count`. This is the total number of GCs *since the process booted*, not that ran during the profiling run.

Each GC run appears on its own line, with some useful, self-explanatory details. The "Total Size" is the current size of the Ruby heap in bytes.

When to use GC and GC::Profiler: If one of your other tools points you towards garbage collection occurring often or taking a long time, `GC` and `GC::Profiler` are your friend. You'll be able to see just how often garbage collection runs in a certain block of code or during a request, and how long it takes. Once you've zeroed in on code that causes garbage collection, use the other tools here to figure out why so many objects are being allocated.

gc_tracer

`gc_tracer` was written by Koichi Sasada, Ruby core member. In many ways, it's an extension of `GC::Profiler`, which can log GC information to files and, when mounted to your application, even gives you a webpage with GC information.

To add `gc_tracer` to Rails app, we add it to our Gemfile like so:

```
gem 'gc_tracer', require: 'rack/gc_tracer'
```

Then, we need to insert `gc_tracer`'s middleware. Open up `config.ru`, and insert the following line above `run MyApp::Application`:

```
use Rack::GCTracerMiddleware, view_page_path: '/gc_tracer', filename: 'log/gc_tracer'
```

Now, navigate to `http://localhost:3000/gc_tracer` and you'll get some highly detailed information in a tabular format:

| type | tick | count | heap_allocated_pages | heap_sorted_length | heap_allocatable_pages |
|-----------|------------------|-------|----------------------|--------------------|------------------------|
| end_mark | 1452612270897648 | 37 | 1367 | 1368 | 0 |
| end_sweep | 1452612270951614 | 37 | 1367 | 1368 | 0 |
| start | 1452612270975699 | 38 | 1367 | 1368 | 0 |
| end_mark | 1452612270988874 | 38 | 1368 | 2460 | 1092 |
| end_sweep | 1452612271591575 | 38 | 1921 | 2460 | 538 |
| start | 1452612271634070 | 39 | 1921 | 2460 | 538 |
| end_mark | 1452612271665333 | 39 | 1921 | 2460 | 538 |

You should also have a text file being logged to in the log folder, which can be loaded into Excel.

The output here is fairly opaque - but most of the columns are just a steady log of `gc.stat`, which is something I talk about above.

If you want GC information about background jobs, I recommend using the block format in your job class:

```
GC::Tracer.start_logging do
  # do something
end
```

The log info will go to STDERR. You can also provide a filename to output to a file, see the project's README for more details.

When to use `gc_tracer`: In development, you may want a constant log available of garbage collections and what's causing them. This is exactly what `gc_tracer` provides. `gc_tracer` may be particularly useful in tracking down memory leaks.

derailed_benchmarks

`derailed_benchmarks` is a complete benchmarking suite for Rails written by Richard Schneeman. It's a great project, and while it has lots of swiss-army-knife features for benchmarking all areas of a Rails app, I want to talk about its memory benchmarks. I know I said this would be a profiling lesson, but bear with me!

My favorite feature of `derailed_benchmarks` is for tracking down memory bloat. You've got an app that runs at 512MB of RAM or more on production, where do you start? I start at the Gemfile and fire up `derailed_benchmarks`. Add `derailed` to the development section of your application's Gemfile and run `bundle exec derailed bundle:mem`. Wait a second while `derailed` requires each of your gems individually and checks to see how much memory they use.

Here's some example output:

```

delayed_job: 18.9805 MiB (Also required by: delayed/railtie, delayed_job_active_re
cord)
  delayed/performable_mailer: 17.8633 MiB
    mail: 17.8555 MiB (Also required by: TOP)
      mime/types: 12.9492 MiB (Also required by: /Users/nateberkopec/.gem/ruby/2
.3.0/gems/rest-client-1.8.0/lib/restclient/request, /Users/nateberkopec/.gem/ruby/
2.3.0/gems/rest-client-1.8.0/lib/restclient/payload)
        mail/field: 2.0039 MiB
        mail/message: 0.8477 MiB
      delayed/worker: 0.6055 MiB
    rails/all: 15.8125 MiB
      rails: 7.5352 MiB (Also required by: active_record/railtie, active_model/railt
ie, and 10 others)
        rails/application: 5.3867 MiB
[... continues on and on]

```

The indentations in the output show dependencies - `delayed_job` requires `delayed/performable_mailer` and so on.

In this output, we can see that `mime/types` is required by `delayed_job` and several other libraries. The `mime-types` gem had a huge memory bloat problem that Richard fixed a while ago, but most gem authors haven't updated their `.gemspec` files to require the correct version of the `mime-types` gem that fixes this problem.

This `mime-type` bloat is an extremely common problem in Ruby gemfiles. Go check your apps right now - I bet at least one of them has this issue.

To fix it, you can try:

```
gem 'mime-types', '~> 3.0'
```

...at the *top* of your Gemfile. If you get a version conflict (you use a gem that depends on `mime-types` version 2, you can do this:

```
gem 'mime-types', '~> 2.6', require: 'mime/types/columnar'
```

Relax, sit back, and enjoy 20MB of free extra memory. Rinse and repeat this process for other gems - look for files that require a lot of memory and find ways to eliminate them or use less heavyweight alternatives. Swap `carrierwave` for `carrierwave-aws`, and so on.

While `derailed bundle:mem` is a static benchmark (it never actually fires up your app), `derailed` also has several dynamic benchmarks that actually boot your app and `curl` some requests against it to simulate user load.

There are three dynamic memory benchmarks in `derailed` that I find useful:

- `derailed exec perf:mem_over_time` hits your app a bunch of times and outputs total process memory. If the number keeps increasing, congratulations, you've got a leak. If it levels off, you're good.
- `derailed exec perf:objects` hits your app and looks to see where objects are created. Use it to pinpoint the memory-expensive operations in your application.

All of these dynamic benchmarks use the `PATH_TO_HIT` and `TEST_COUNT` environment variables to determine what path to test and how many times to test it. For example, to run a benchmark 10 times against `/signup`:

```
PATH_TO_HIT=/signup TEST_COUNT=10 derailed exec perf:objects
```

When to use `derailed`: On any app, really. `bundle:mem` is excellent for auditing Gemfiles and reducing bloat. The dynamic benchmarks can help track down memory leaks.

memory_profiler

`memory_profiler` was originally written by Sam Saffron, tech head of Discourse. It's used under-the-hood by `derailed`, but it's often useful to use `memory_profiler` alone.

One reason `memory_profiler` is useful is that it can be used like a traditional profiler and only profile a block of code:

```
require 'memory_profiler'
report = MemoryProfiler.report do
  # run your code here
end

report.pretty_print
```

This is useful when debugging memory-heavy backend jobs, something which `derailed` isn't designed for.

`memory_profiler`'s reports are extensive and a little overwhelming at first.

```
Total allocated: 914156 bytes (8503 objects)
Total retained: 46834 bytes (645 objects)
```


At the top of the report, `memory_profiler` reports the total amount of memory allocated and retained while it was run. *Retained memory* is memory used by objects which will live on beyond the profiler run - these are objects which survived garbage collection (so far). *Allocated memory* is all memory allocated during the profiler run - this can be higher than your maximum memory usage. Consider - Ruby allocates 20 kB of memory, then GC runs and frees up 10 kB. Then, Ruby allocates an additional 5kB of memory.

`memory_profiler` will report 25000 bytes allocated, but your maximum memory usage (at the end of the run) will be just 20000 bytes. A high amount of allocated memory is still an important metric, however - more memory allocated means the garbage collector will run more often, slowing down your process.

All the numbers in `memory_profiler` are in bytes, but it's important to note that they don't reflect total process memory usage. Due to memory fragmentation over time,

`memory_profiler` will always underestimate versus what you might see if you checked your Ruby process' memory usage with `ps`.

You can use `memory_profiler` by itself to profile blocks of code, but `memory_profiler` is generally easiest to use when you use it with other gems. `rack-mini-profiler` uses

`memory_profiler` to profile memory during a request/response in a Rack app, and `derailed` (above) uses it for several benchmarks.

`memory_profiler` also works with C extensions, so it will correctly profile memory usage of gems like Nokogiri that use C extensions for significant functionality.

When to use `memory_profiler`: Use `memory_profiler` when debugging memory usage of background jobs or other non-Rack-app scenarios. If debugging memory usage of a Rack app, use `derailed` and `rack-mini-profiler`, which employ `memory_profiler` as a backend.

Checklist for Your App

- Perform an audit of your Gemfile with `derailed_benchmarks`. Substitute or eliminate bloated dependencies - `derailed`'s "TOP" output should probably be 50-60 MB.
- Experiment with `ObjectSpace` by writing a logger for your application that tracks areas you suspect may be memory hotspots.
- If you're using `rack-mini-profiler`, install `memory_profiler` to enable RMP's memory profiling functions.

Lab: Memory Profiling

Exercise 1

Using `Rubygems.org` (setup described in another lesson), profile memory usage on boot with `memory_profiler` .

Add `memory_profiler` to the Gemfile, then change `environment.rb` to look like the following:

```
MemoryProfiler.report do
  require File.expand_path('../application', __FILE__)
  Rails.application.initialize!
end.pretty_print(to_file: "my_report.txt")
```

Run `rails runner "puts 'hello world'"` in your console, and open `my_report.txt` .

What conclusions can you draw about the memory usage of this application?

Solution

- ActiveSupport::Dependencies requires a lot of memory - this module is what keeps track of your files for reloading in development.
- The `tzinfo` gem requires a decent amount of memory to run - check out `zoneinfo_data_source.rb` on line 367 and `country_timezone.rb` on line 32.

rack-mini-profiler - the Secret Weapon

`rack-mini-profiler` is a performance tool for Rack applications, maintained by the talented [@samsaffron](#). `rack-mini-profiler` provides an entire suite of tools for measuring the performance of Rack-enabled web applications, including detailed drill downs on SQL queries, server response times (with a breakdown for each template and partial), incredibly detailed millisecond-by-millisecond breakdowns of execution times with the incredible `flamegraph` feature, and will even help you track down memory leaks with its excellent garbage collection features. **I wouldn't hesitate to say that `rack-mini-profiler` is my favorite and most important tool for developing fast Ruby webapps.**

The best part - `rack-mini-profiler` is designed to be run in production. Yeah! You can accurately profile production performance (say that three times fast) with `rack-mini-profiler`. Of course, it also works fine in development. But your development environment is usually a lot different than production - hardware, virtualization environments, and system configuration can all be different and play a huge part in performance. Not to mention Rails' development mode settings, like reloading classes on every request!

In this post, I'm going to take a deep dive on `rack-mini-profiler` and show you how to use each of its powerful features to maximize the performance of your Rails app.

Installation

For the purposes of this demo, I'm going to assume you're in a Rails app. The installation procedure is slightly different for a pure Rack app, [see the README for more](#).

First, let's add the following gems to our Gemfile, below any database gems like 'pg' or 'mysql2'.

```
gem 'pg' # etc etc

gem 'rack-mini-profiler'
gem 'flamegraph'
gem 'stackprof' # ruby 2.1+ only
gem 'memory_profiler'
```

`rack-mini-profiler` is self explanatory, but what are the other gems doing here?

`flamegraph` will give us the super-pretty flame graphs that we're going to use later on.

`stackprof` is a stack profiler (imagine that), which will be important when we start building our flame graphs. This gem is Ruby 2.1+ only - don't include it otherwise (`rack-mini-profiler` will fallback to another gem, `fast_stack`).

`memory_profiler` will let us use `rack-mini-profiler` 's GC features.

Fire up a server in development mode and hit a page. You should see the new speed

badge in the upper left.  We'll get to what that does in a second.

To see a full list of `rack-mini-profiler`'s features and info on how to trigger them, add `?pp=help` to the end of any URL.¹

We're going to go through all of these options - but first, we need to make our app run in production mode on our local machine.

`rack-mini-profiler` is designed to be used in production. In Rails, your application probably behaves differently in production mode than in development mode - in fact, most Rails apps are 5-10x slower in development than they are in production, thanks to all the code reloading and asset recompilation that happens per request. So when profiling for speed, run your server in production mode, even when just checking up on things locally. Be careful, of course - change your `database.yml` file so that it doesn't point towards your *actual* production database (not necessary for Heroku-deployed apps).

`rack-mini-profiler` runs in the development environment by default in Rails apps. We're going to enable it in production, and hide it behind a URL parameter. You can also do things like make it visible only to admin users, etc.

```
# in your application_controller

before_filter :check_rack_mini_profiler
def check_rack_mini_profiler
  # for example - if current_user.admin?
  if params[:rmp]
    Rack::MiniProfiler.authorize_request
  end
end
```

Also, I prefer not to use `rack-mini-profiler`'s default storage settings in production. By default, it uses the filesystem to store data. This is slow to begin with, and especially slow if you're on Heroku (which doesn't have a real filesystem).

```
# in an initializer
Rack::MiniProfiler.config.storage = Rack::MiniProfiler::MemoryStore
```

If you're forcing SSL in production, you're going to want to turn that off for now.

```
config.force_ssl = false
```

Finally, I need to get the app running in production mode.² In my case (a Rails 4.2 app), I just have to run the database setup tasks in production mode, compile assets, and add a secret key base to my rails server command:

```
RAILS_ENV=production rake db:reset # CAREFUL!
RAILS_ENV=production rake assets:precompile
RAILS_ENV=production SECRET_KEY_BASE=test rails s
```

The Speed Badge

Great - you've got the speed badge. In my example app, starting the rails server in development mode and then hitting the root url actually causes two speed badges to show up. `rack-mini-profiler` will create a speed badge for each request made to your app, including some asset requests. In my case, I also got a speed badge for the favicon request.

When you click on the speed badge, you can see that `rack-mini-profiler` breaks down the time your page took to render on a per-template basis. It breaks out execution time spent in the layout, for example, and then break out each partial that was rendered as well. Here's an example readout from a different app I work on:

| localhost on Tue, 28 Jul 2015 14:58:45 GMT | | | | |
|--|--------------|-----------------|-----------------|---------|
| | content (ms) | from start (ms) | query time (ms) | |
| GET http://localhost:5000/ | 10.6 | +0.0 | 3 sql | 0.9 |
| Executing action: home | 9.6 | +4.0 | 2 sql | 1.4 |
| Rendering: landings/home | 6.2 | +13.0 | 1 sql | 0.5 |
| Rendering: landings/_dynamic_slider | 16.9 | +16.0 | 1 sql | 0.7 |
| Rendering: blog/posts/_post | 6.8 | +34.0 | | |
| Rendering: blog/posts/_post | 6.1 | +41.0 | | |
| Rendering: blog/posts/_post | 16.9 | +47.0 | 2 sql | 1.3 |
| Rendering: layouts/application | 1.7 | +66.0 | | |
| Rendering: layouts/_header | 5.8 | +68.0 | | |
| show time with children | | | 5.7 | (16.3%) |
| | time (ms) | from start (ms) | | |
| Response | 3.0 | +113.0 | | |
| Dom Content Loaded Event | 69.0 | +364.0 | | |
| First Paint Time | | +512.0 | | |
| Load Event | 16.0 | +720.0 | | |
| share | | | show trivial | |

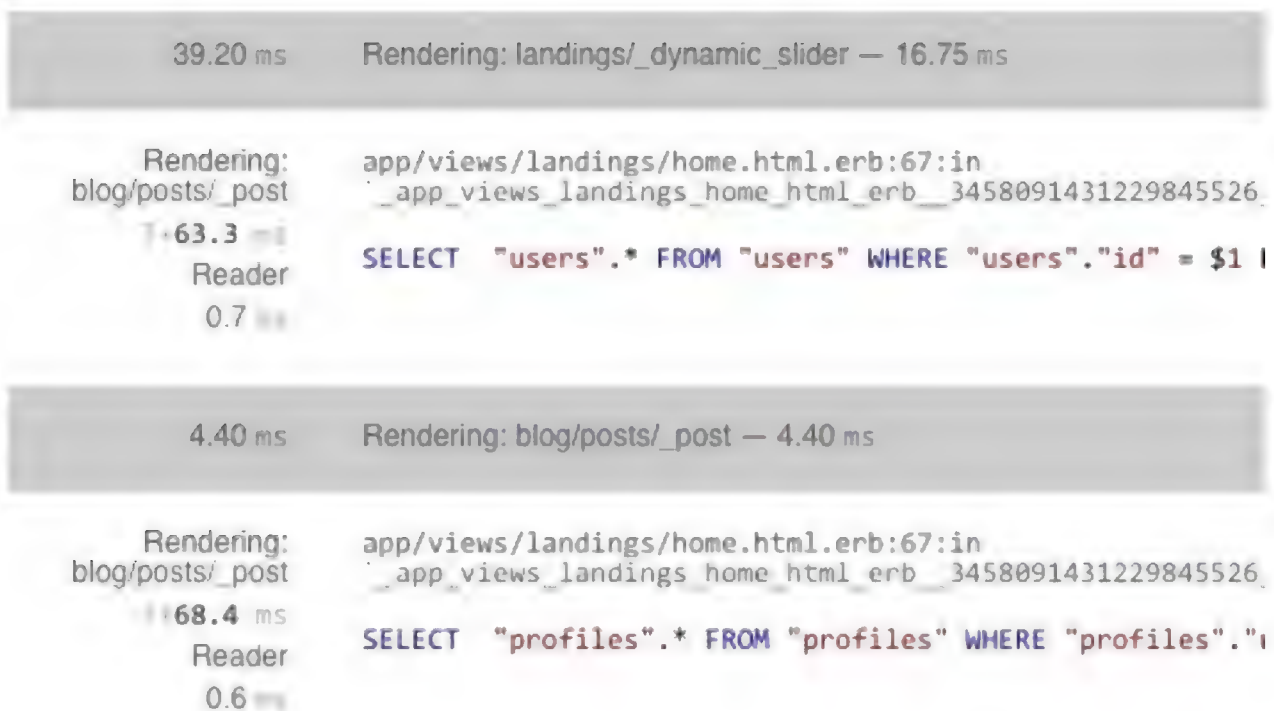
I think this view is pretty self explanatory so far. You're looking at exactly where your time goes on each request in a brief overview. When I look at this view for any given request, here's what I look for:

- *How many SQL queries am I generating?* This view generates a total of 9 SQL queries. That strikes me as lot, especially since this is just the homepage for a non-logged-in user. Usually, for simple pages, you wouldn't want to see more than 1 to 3 queries, and almost always you'd like just one query per ActiveRecord model class.
- *What's my total request time?* This view is a little slow - 85ms. For a mostly static and highly visited page like this (like I said, it's the homepage) I'd like to see it be completed in under 50ms.
- *What % of time am I spending in SQL?* This view is doing fairly well as far as time spent in SQL goes. I always test my applications with a copy of the production database - this makes sure that my query results match production results as much as possible. Too often, simplistic development databases return 1000 results where a production database would return 100,000.
- *How long until DOMContentLoaded fires?* This view took about 250ms between receiving a response and finishing loading all the content. That's pretty good for a simple page like this. Decreasing this time requires front-end optimization - something I can't get into in this post, but doing things like reducing the number of

event handlers and front-end JavaScript, and optimizing the order of external resources being loaded onto the page.

- *Are any of the parts of the page taking up an extreme amount of time compared to others?* Sometimes, just a single partial is taking up the majority of the page load time. If that's true, that's where I start digging for more information. In this case, the page's load time looks fairly evenly distributed. It looks like one of the post partials here is generating some SQL - a prime candidate for caching (or just getting rid of the query in the first place).

There are some other features here in the speed badge. Click any of the SQL links and you'll see the exact query being executed. Here are two as an example:



The number on the top left (39.20 ms) is the total time spent between rendering this partial and the next one - notice that this is slightly different than the number to the right, the amount of time actually spent rendering the partial (16.75ms). Whenever I see "lost time" like this, I dig in with the flamegraph tool to see exactly where the time went. We'll get into that in the next section.

Notice that `rack-mini-profiler` calls out the exact line in our view that triggered the query.

These queries look like the view was probably looking up the `current_user` (or some other user), and that `current_user` has one `Profile`. I probably need to:

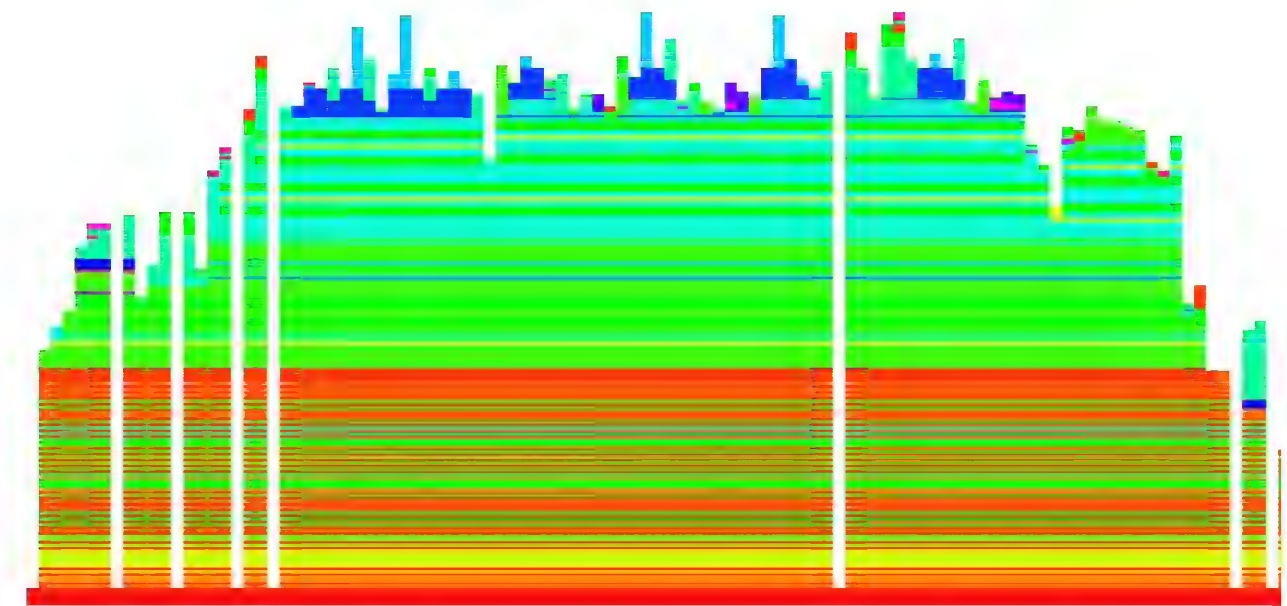
- Find a way to either eliminate this query or cache the result in the view

- Add an `includes` call to the original query so that the profile is loaded along with the User, reducing my query count by 1.

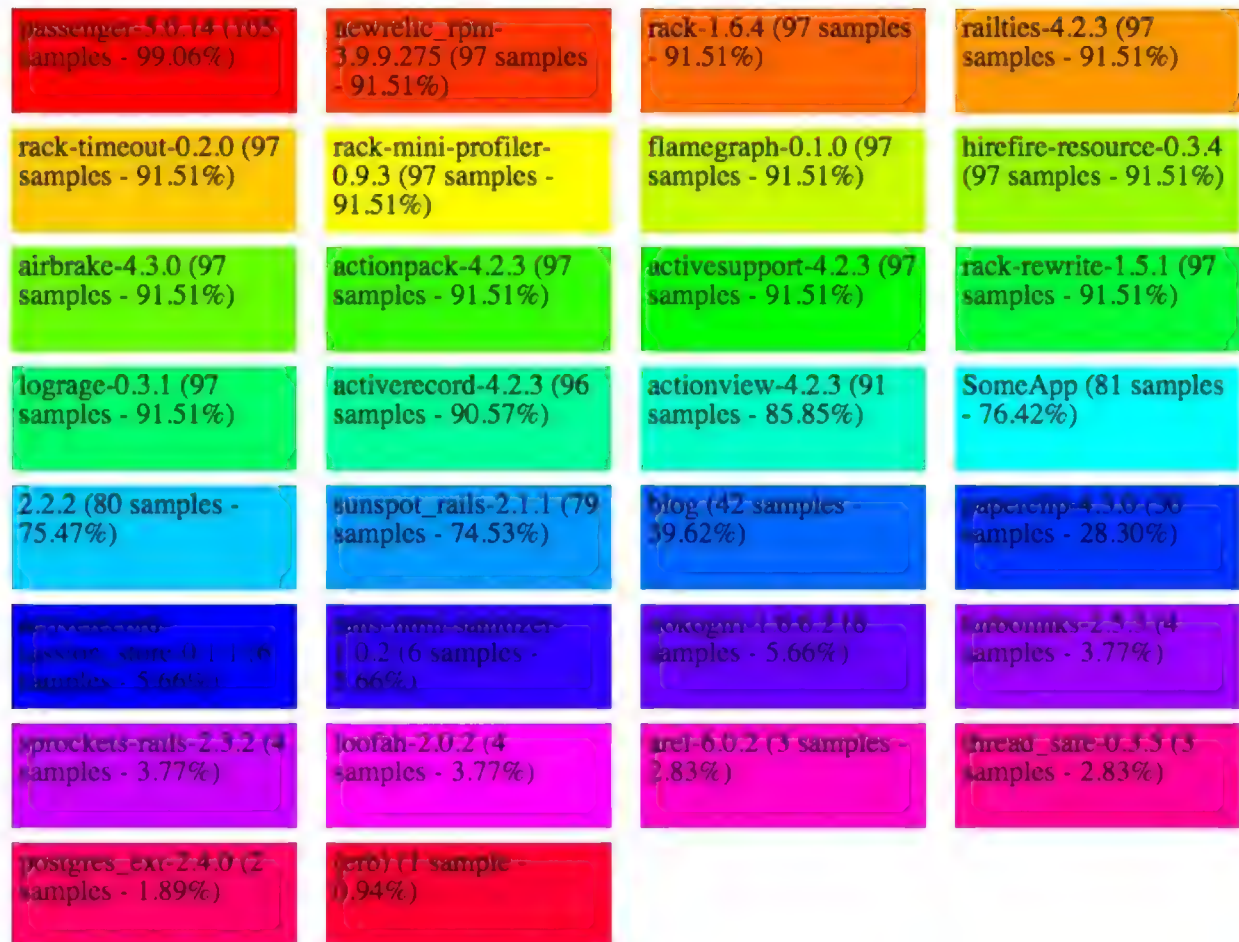
I follow this process for every query on the page - see if I can remove it or cache the result. [For my full guide on Rails caching, check this post out.](#)

The Flamegraph

This is one of my favorite parts of `rack-mini-profiler`, and as far as I know, not duplicated anywhere else. If I add `?pp=flamegraph` to my query string, I can get this incredible flamegraph of the same request I outlined above:



The height of the "flame" indicates how deep we are in the stack. Think of the Y axis as stack level, and the X axis as time. You can zoom in and out with your mouse scroll wheel.



At the bottom of the page, you'll see a legend, denoting what all the colors refer to. Note that the percentage displayed next to each part is the *percentage of the time the request spent inside that stack frame*. For example, this app is called SomeApp. It looks like we spent 76.42% of our time in the app itself. The other time was taken up by rack middleware (like `lograge`, `airbrake` and `hirefire-resource`) and Rails.

Looking at this legend and poking around the flamegraph reveals an interesting tidbit - Paperclip appeared in 28.3% of all stack frames! Yikes. That's way too many for a simple homepage. For this page, I'd look into ways of avoiding calls to Paperclip. It looks like most of the time is spent generating a `Paperclip::Attachment's URL`. I may experiment with ways to cache or otherwise avoid recalculating that value.

GC Profiling

Here's another awesome part of `rack-mini-profiler` that I haven't seen anywhere else - a set of tools for debugging memory issues *live on production!* Even better, it incurs no performance penalty for requests where `rack-mini-profiler` is not enabled!

profile-gc

So let's add `pp=profile-gc` to our query string and see what we get (the output is usually enormous and takes a while to generate):

```
Overview
-----
Initial state: object count - 331594, memory allocated outside heap (bytes) 75806422

GC Stats: count : 39, heap_allocated_pages : 1792, heap_sorted_length : 2124, heap_
_allocatable_pages : 353, heap_available_slots : 730429, heap_live_slots : 386538,
 heap_free_slots : 343891, heap_final_slots : 0, heap_marked_slots : 386536, heap_
swept_slots : 343899, heap_edén_pages : 1439, heap_tomb_pages : 353, total_allocat
ed_pages : 1852, total_freed_pages : 60, total_allocated_objects : 4219050, total_
freed_objects : 3832512, malloc_increase_bytes : 960, malloc_increase_bytes_limit
: 26868266, minor_gc_count : 27, major_gc_count : 12, remembered_wb_unprotected_ob
jects : 9779, remembered_wb_unprotected_objects_limit : 19558, old_objects : 36615
6, old_objects_limit : 732312, oldmalloc_increase_bytes : 1344, oldmalloc_increase
_bytes_limit : 22319354

New bytes allocated outside of Ruby heaps: 1909904
New objects: 17029
```

Here's the first section. If that output looks familiar to you, it is - it's the output of `GC.stat`. [GC is a module from the stdlib](#) that has a whole lot of convenience methods for working with the garbage collector. `stat` gives us that output above. For a full explanation about what each of those values mean, read Sam's post [on how Ruby's GC works](#).

At the bottom, you'll see the new bytes allocated outside of Ruby heaps, along with a count of new objects. Pay attention to any requests that generate abnormally high values here (10+ MB allocated per request, for example).

Here's the next section:


```
ObjectSpace delta caused by request:
```

```
-----
String : 9285
Array : 3641
Hash : 1421
Regexp : 375
MatchData : 349
RubyVM::Env : 214
Proc : 204
Time : 173
Psych::Nodes::Scalar : 168
...
```

This section shows us the change (that's what delta means) in the total objects in the ObjectSpace that the request caused. For example, after the request, we have 9285 more Strings than before.

`ObjectSpace` is an incredibly powerful module - for example, with

`ObjectSpace.each_object` you can iterate through *every single currently allocated object in the Ruby VM*. That's insane!

I don't find this section useful on its own - though a huge number of app-specific objects (for example, let's say 2,000 `Paperclip::Attachment` s) may be a red flag.

```
ObjectSpace stats:
```

```
-----
String : 175071
Array : 49440
RubyVM::InstructionSequence : 32724
ActiveSupport::Multibyte::Unicode::Codepoint : 27269
Hash : 12748
RubyVM::Env : 8102
Proc : 7806
MIME::Types::Container : 3816
Class : 3371
Regexp : 2739
MIME::Type : 1907
...
```

Here's the *total* number of Objects, by Class, alive in the VM. This one is considerably more interesting for my application. What's with all of those `MIME::Type` s and

`MIME::Types::Container` objects? I suspect it might have something to do with Paperclip,

but then again, nearly every gem uses MIME types somehow. In fact, it's such a notorious memory hog that [Richard Schneeman recently saved roughly 50,000 objects from being created with just a single change!](#)

```
String stats:
-----
444 :
352 : :
218 : /
129 : :s3_path_url
117 :
116 :

108 : a
106 : href
96 : <<
78 : [&"'><]
78 : index
73 : # Amazon S3 Credentials

...
```

Here's the final bit of output - a count on the number of times a certain string was allocated. For example, the string "index" has been allocated 78 times.

This output is useful to determine if a string should be extracted to a constant and frozen. For example, [this is what Rack does here with the string "chunked"](#).

Why would we do this? If, for example, Rack was allocating the string "chunked" 1000 times in a single request, we can reduce that to 1 time by only referring to a constant value. [In fact, that's exactly why this was done.](#)

If all of this memory stuff is going over your head, don't worry. I recommend watching [John Crepezzi's talk On Memory](#) for an intro to how memory works in Ruby.

profile-memory

The `pp=profile-memory` parameter uses the excellent `memory_profiler` gem (which you should use on its own to benchmark other code). It's like a hopped-version of `profile-gc` from earlier. Instead of just telling us *what* Strings were allocated during a request, `profile-cg-ruby-head` tells us exactly *what line of code allocated that String*. This is *extremely powerful*.

Here's some example output:


```
Total allocated 16986
```

```
Total retained 1208
```

```
allocated memory by gem
```

```
-----
 769864  paperclip-4.3.0
 382958  activesupport-4.2.3
 324621  actionpack-4.2.3
 274792  activerecord-4.2.3
 246966  2.2.2/lib
 234562  actionview-4.2.3
 118650  newrelic_rpm-3.9.9.275
  72424  rack-1.6.4
  69359  nokogiri-1.6.6.2
  43845  SomeApp/app
  ....
```

```
allocated memory by file
```

```
-----
 689672  ~/gems/paperclip-4.3.0/lib/paperclip/interpolations.rb
 224356  ~/gems/activesupport-4.2.3/lib/active_support/core_ext/string/output_s
afety.rb
 136744  ~/gems/actionpack-4.2.3/lib/action_dispatch/routing/route_set.rb
 104800  ~/.rbenv/versions/2.2.2/lib/ruby/2.2.0/erb.rb
  84291  ~/gems/actionview-4.2.3/lib/action_view/helpers/tag_helper.rb
  76272  ~/gems/actionpack-4.2.3/lib/action_dispatch/journey/formatter.rb
  53964  ~/gems/activerecord-4.2.3/lib/active_record/connection_adapters/postgr
esql_adapter.rb
  52145  ~/gems/rack-1.6.4/lib/rack/response.rb
  43824  ~/.rbenv/versions/2.2.2/lib/ruby/2.2.0/psych/scalar_scanner.rb
  ....
```

```
allocated objects by gem
```

```
-----
  4321  paperclip-4.3.0
  2322  activerecord-4.2.3
  2300  actionpack-4.2.3
  2082  actionview-4.2.3
  1726  activesupport-4.2.3
  1538  2.2.2/lib
   981  newrelic_rpm-3.9.9.275
```

There's Paperclip again! Note that this output of the first section (allocated memory) is in bytes, which means Paperclip is allocating about 1 MB of objects for this request. That's a lot, but I'm not quite worried yet. But this view in general is a good way of finding memory hogs. The actual RAM cost will always be slightly higher than what is reported here. MRI heaps are not squashed to size.

Oh - and what does "allocated" mean, exactly? `memory_profiler` differentiates between an "allocated" and a "retained" object. A "retained" object will live on beyond this request, probably at *least* until the next garbage collection. It may or may not be garbage collected at that time.

An allocated object may or may not be retained. If it isn't retained, it's just a temporary variable that Ruby knows to throw away when it's done with. Retained objects are ones we should really worry about though - which is contained later on in the report.

Keep scrolling down and you'll see the same output, but for "retained" objects only. Pay attention in this area - all of these objects will stick around after this request is over. If you're looking for a memory leak, it's in there somewhere.

analyze-memory

`pp=analyze-memory`, new with `rack-mini-profiler` version 0.9.6, performs some basic heap analysis and lists the 100 largest strings in the heap. Usually, the largest one is your response.

I haven't found a lot of use for this view either, but if you're tracking down String allocations, you may find it useful.

Exception Tracing

Did you know that raising an Exception in Ruby is slow? [Well, it is. Up to 32x slower.](#) And unfortunately, *some people* and *certain gems* use exceptions as a form of flow control. For example, the `stripe` gem for Ruby raises an Exception when a credit card transaction is denied.

Your app should not raise Exceptions anywhere during normal operation. Your libraries may be doing this (and of course, catching them) without your knowledge. If you suspect you've got a problem with exceptions being raised and caught in your stack, give

```
pp=trace-exceptions a try.
```

Conclusion

That wraps up our tour of `rack-mini-profiler`. I hope you've enjoyed this in-depth tour of the Swiss army knife of Rack/Ruby performance.

Checklist for Your App

- Set up `rack-mini-profiler` to run in production.
- Set up your application so it can run in production mode locally, on your machine.
- Use `rack-mini-profiler` to see how many SQL queries pages generate in your app. Are there pages that generate more than dozen queries or so, or generate several queries to the same table?
- Use `rack-mini-profiler 's' trace-exceptions` feature to make sure you aren't silently raising and catching any exceptions.

¹. In more recent versions of `rack-mini-profiler`, there's also a 'help' button on the speed badge" - this prints the help screen and lists the various commands available (all used by adding to the URL query string) ↩

². Also, if you're having trouble getting the speed badge to show up in production mode and you're using `Rack::Deflater` or any other gzipping middleware, [you need to do some other stuff](#) to make sure `rack-mini-profiler` isn't trying to insert HTML into a gzipped response. ↩

Lab: rack-mini-profiler

This lab requires that you have pulled down and set up `Rubygems.org`. See `RUBYGEMS_SETUP.md` if you have not already set up the application locally.

Exercise 1

Add `rack-mini-profiler` to `Rubygems.org` and try using it to identify some performance issues. What opportunities for improvement do you see? Concentrate on problems that are exposed by `rack-mini-profiler`, with a focus on response time speeds. Click around and look at the different pages, not just the homepage. Note that "GUIDES" and "CONTRIBUTE" actually take you to a separate site.

Be sure to use the flamegraph and other features of `rack-mini-profiler` !

Solution

- The "announcements" system triggers a SQL query on every page. The result of this query should be cached to prevent this query from occurring on every page.

Creating a new Announcement would bust this cache.

- The "gems" page (RubygemsController#index) contains an N+1 query for fetching `version` objects. This is also true of the search page.
- Searching is slow. The search page lacks the correct index - gems should be indexed using their uppercase names with "UPPER()", like the search query uses. For more about this, see the databases lesson.

There are probably other opportunities that I didn't identify - please let me know if you think there are obvious ones that I've missed.

Performance Monitoring with New Relic

It's 12 p.m. on a Monday. Your boss walks by: "The site feels...slow. I don't know, it just does." Hmmm. You riposte with the classic developer reply: "Well, it's fast on my local machine." Boom! Boss averted!

Unfortunately, you were a little dishonest. You know better than to think that speed in the local environment has anything to do with speed in production. You know that, right? Wait, we're not on the same page here?

Several factors can cause Ruby applications to have performance discrepancies between production and development:

- **Application settings, like code reloading** Rails and most other Ruby web frameworks reload (almost) all of your application code on every request to pick up on changes you've made to files. That's a pretty slow process. In addition, there a lot of subtle differences between apps in development and production modes, especially surrounding asset pipelines. Simpler frameworks may not have these behaviors, but don't kid yourself - if anything in your app is changing in response to "RACK_ENV", you could be introducing performance problems that you can't catch in development.
- **Caching behavior** Rails disables caching in development mode by default. Turning that on has a big performance impact in production. In addition, caches in development work differently than caches in production, mostly due to the introduction of network latency. Even 10ms of network latency between your application server and your cache store can cripple pages that make many cache calls.
- **Differences in data** This is an insidious one, and the usual cause for an app that seems slow in production but fast in development. Sure, that query you run locally (User.all, for example) only returns 100 rows in development using your seed data. But in production, that query could return 10,000 or 100,000 rows! In addition, consider what happens when those 10,000 rows you return need to be attached to 10,000 other records because you used `includes` to pre-load them. Get ready to wait around.
- **Network latency** It takes time for a TCP packet to go from one place to another. And while the speed of light is fast, it does add up. As a rule of thumb, figure 10ms for the same city, 20ms for a state or two away, 100ms across the US (from NY to

CA), and up to 300ms to get to the other side of the world. These numbers can quadruple on mobile networks. This has a major impact when a page makes many asset requests or has blocking external JavaScript resources.

- **JavaScript and devices** JavaScript takes time and CPU to execute. Most people don't have fancy new MacBooks like we developers do - and on mobile devices the story is certainly even worse. Consider that even a low-end desktop processor sports twice the computing power of top-end mobile CPUs and you can see how complex Javascript that "feels fine on my machine" can grind a mobile device to a halt.
- **System configuration and resources** Unless you're using containers, system configuration will always differ between environments. This can even be as subtle as utilities being compiled with different compiler flags! Of course, even containers will run on different physical hardware, which can have severe performance consequences, especially regarding threading and concurrency.
- **Virtualization** Most people deploy to shared, virtualized environments nowadays. Unfortunately, that means a physical server will share resources with up to half-a-dozen or so virtual servers, which can negatively and unpredictably impact performance when one virtualized server is hogging up the resources available.

So what's a developer to do? Why, install a performance monitoring solution in production! NewRelic is the tool I reach for. Not only is it free to start with, the tools included are extensive, even at the free level. In this post, I'm going to give you a tour of each of NewRelic's features and how they can help you to diagnose performance hotspots in a Rails app.

Full disclosure - I don't work for New Relic, and no one from New Relic paid for or even talked to me about this lesson.

Getting an Overview

Let's walk through the process I use when I look at a Ruby app on NewRelic.

When I first open up a New Relic dashboard, I'm trying to establish the broad picture: How big is this application? Where does most of its time go? Are there any "alarm bells" going off just on the main dashboard?

A Glossary

New Relic uses a couple of terms that we'll need to define:

- **Transactions** This is New Relic's cross-platform way of saying "response". In Rails, a single "transaction" would be a single response from a controller action. Transactions from a Rails app in NewRelic look like "WelcomeController#index" and so on.
- **Real-User Monitoring (also RUM and Browser monitoring)** If you enable it, New Relic will automatically insert some Javascript for you on every page. This Javascript hooks into the [NavigationTimingAPI](#) of the browser and sends several important metrics back to NewRelic. Events set include domContentLoaded, domfomplete, requestStart and responseEnd. Any time you see NewRelic refer to "real-user monitoring" or "browser metrics", they're referring to this.

Response time - where does it go?

The web transaction response time graph is one of the most important on NewRelic, and forms the broadest possible picture of the backend performance of your app. NewRelic defaults to 30 minutes as the the timeframe, but I immediately change this to the longest interval available - preferably about a month, although 7 days will do.

The first thing I'll look at here is the app server and browser response averages. Here are some rules of thumb for what you should expect these numbers to be in an average Rails application:

| App server avg response time | Status |
|------------------------------|---------|
| < 100ms | Fast! |
| < 300ms | Average |
| > 300ms | Slow! |

Of course, those numbers are just rules of thumb for Rails applications that serve up HTML - your typical "Basecamp-style" application. For simple API servers that serve JSON only, I might divide by 2, for example.

| Browser avg load time | Status |
|-----------------------|---------|
| < 3 sec | Fast! |
| < 6 sec | Average |
| > 6 sec | Slow! |

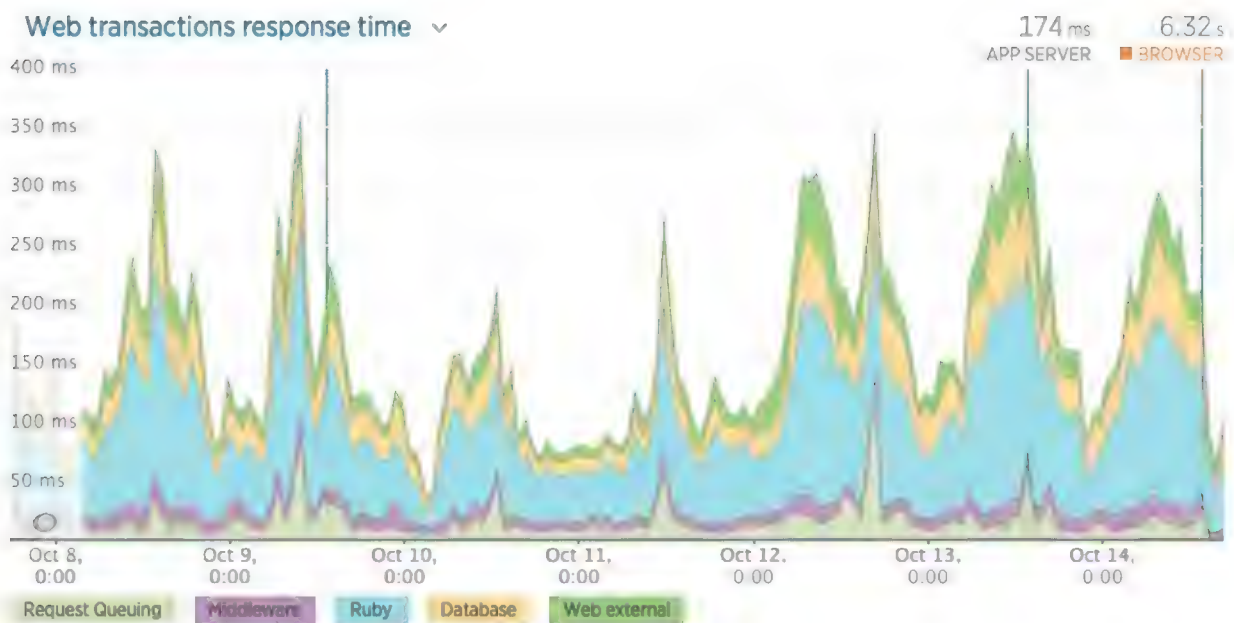
I can hear the keyboards clattering already furiously emailing me: "That's so slow! Rails sucks! Blah blah..."

I'm just sharing what I've seen in the wild in my own experience. Remember - Github, Basecamp and Shopify are all *enormous* WebScale™ Ruby shops that average 50-100ms responses, which is pretty good by anyone's measure.

Based on what I'm seeing with these numbers, I know where to pay attention later on. For example, if I notice a fast or average backend but slow browser (real-user monitoring) numbers, I'll go look at the browser numbers next rather than delving deeper into the backend numbers.

Note that most browser load times are 1-3 seconds, while most application server response times are 1-300 milliseconds. Application server responses, *on average*, are just 10% of the end-users total page loading experience. This means front-end performance optimization is actually far more important than most Rails developers will give it credit for. Back-end optimization remains important for scaling (lower response times mean more responses per second), but when thinking about the browser experience, they usually mean vanishingly little.

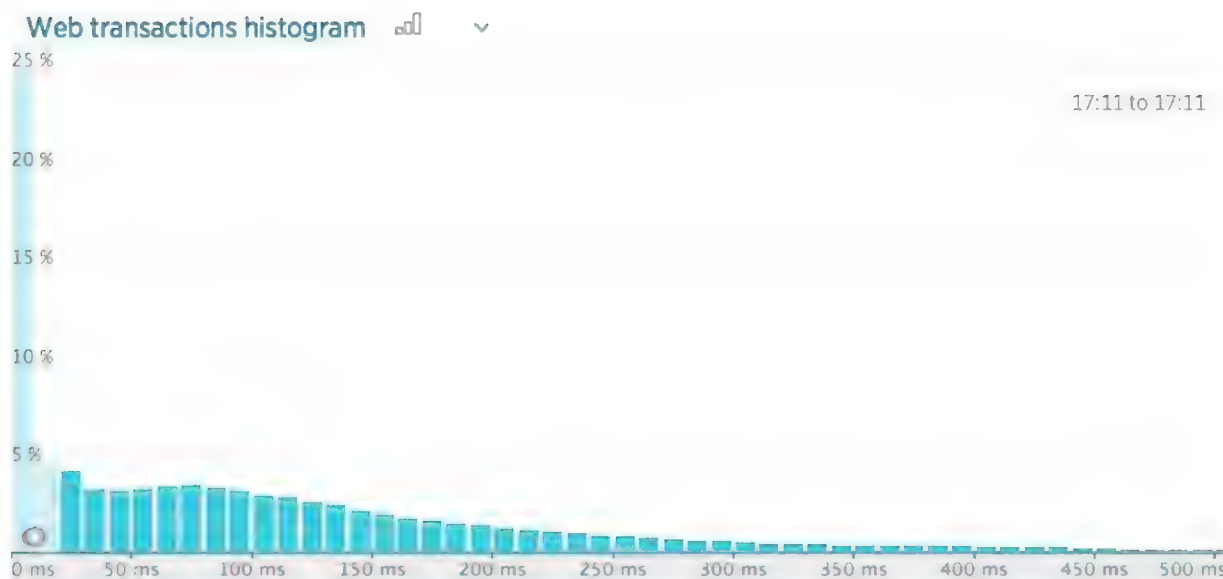
Next, I'm considering the shape of the response time graph. Does the app seem to slow down at certain times of day or during deploys?



The most important part of this graph, though, is to figure out how much time goes to what part of the stack. Here's a typical Ruby application - most of its time is spent in Ruby. If I see an app that spends a lot of time in the database, web external, or other processes, I know there's a problem. Most of your time should be spent in Ruby (running

Ruby code is usually the slowest part of your app!). If, for example, I see a lot of time in web external, I know there's probably a controller or view that's waiting, synchronously, on an external API. That's almost never necessary and I'd work to remove that. A lot of time in request queueing means you need more servers, because requests are spending too much time waiting for an open application instance.

Percentiles and Histograms



The histogram makes it easy to pick out what transactions are causing extra-long response times. Just click the histogram bars that are way far out to the right and pay attention to what controllers are usually causing these actions. Optimizing these transactions will have the biggest impact on 95% percentile response times.

Most Ruby apps response time histograms look like a power curve. Remember what I said above about Pareto. Conversely, be sure to check out what actions take the least amount of time (the histogram bar furthest to the left). Are they asset requests? Redirects? Errors? Is there any way we can *not* serve these requests (in the case of assets, for example, you should be using a CDN)?

What realm of RPM are we playing in?

It's always helpful to check what "order of magnitude" we're at as far as scale. Here are my rules of thumb:

| Requests per minute | Scale |
|---------------------|---|
| < 10 | Tiny. Should only have 1 server or dyno. |
| 10 - 1000 | Average |
| > 1000 | High. "Just add more servers" may not work anymore. |

Apps above 1000 RPM may start running into scaling issues *outside* of the application in external services, such as databases or cache stores. When I see scale like that, I know my job just got a lot harder because the surface area of potential problems just got bigger.

Transactions



Now that I've gotten the lay of the land, I'll start digging into the specifics. We know the averages, but what about the details? At this stage, I'm looking for my "top 5 worst offenders" - where does the app slow to a crawl? What's the 80/20 of time consumed in

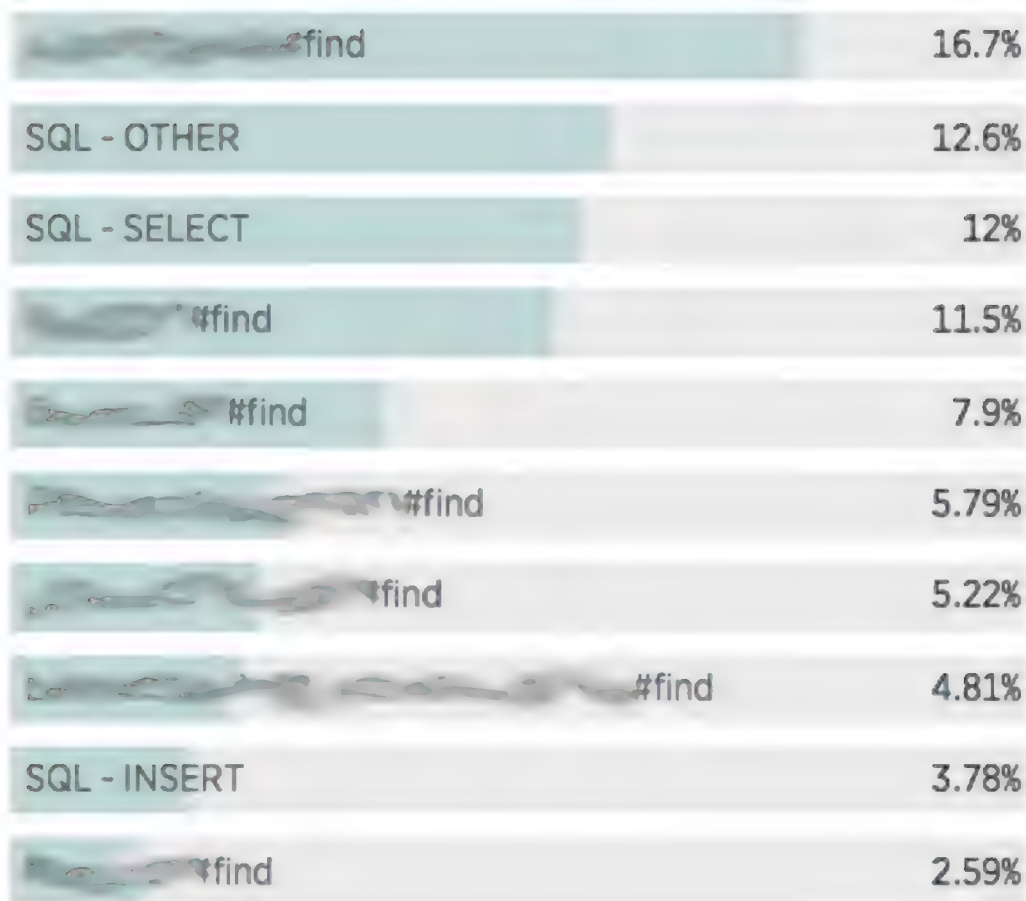
this application - in other words, in what actions does this application spend 80% of its time?

Most Ruby applications will spend 80% of their time in just 20% of the application's controllers (or code). This is good for us performance tweaks - rather than trying to optimize across an entire codebase, we can concentrate on just the top 5 or 10 slowest transactions.

For this reason, in the transactions tab, I almost always sort by *most time consuming*. If the top 5 actions in this tab consume 50% of the server's time (they almost always do), and we speed them up by 2x, we've effectively scaled the application up by 25%! That's free scale.

Alternatively, if an application is on the lower end of the requests-per-minute scale, I might sort by slowest average response time instead. This sort also helps if you're concentrating on squashing 95th percentiles.

Database



I'm carrying

that "worst offender" mindset into the database. Now, if the previous steps have shown that the database isn't a problem, I may glaze over this section or just try and make sure it's not a single query that's taking up all of our database time. Again, "most time consuming" is probably the best sort here.

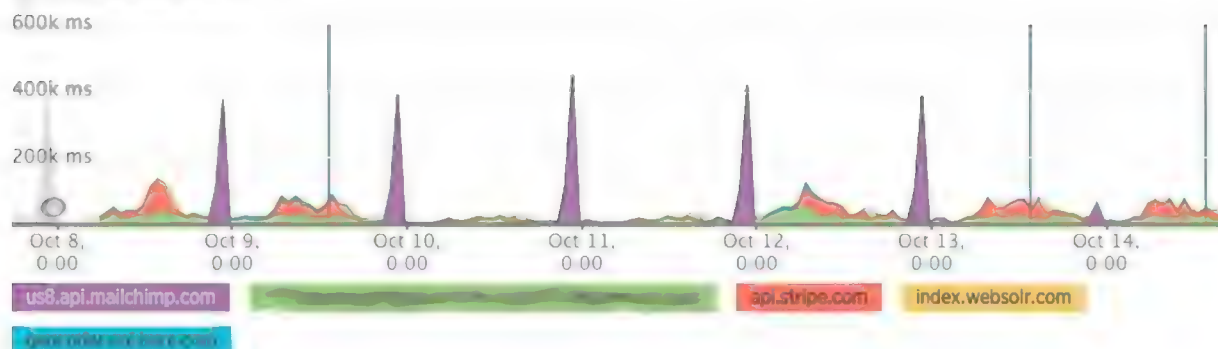
Here's some symptoms you might see here:

- **Lots of time in #find** If your top SQL queries are all model lookups, you've probably got a bad query somewhere. Pay attention to the "time consumption by caller" graph on the right - where is this query being called the most? Go check out those controllers and see if you're doing a WHERE on a column that hasn't been properly indexed, or if you've accidentally added an N+1 query.
- **SQL - OTHER** You may see this one if you've got a Rails app. Rails periodically issues queries just to check if the database connection is active, and those queries show up under this "OTHER" label. Don't worry about them - there isn't really anything you can do about it.

External Services

Top 5 external services

by total response time



What I'm looking for here is to make sure that there aren't any external services being pinged during a request. Sometimes that's inevitable (payment processing) but usually it isn't necessary.

Most Ruby applications will block on network requests. For example, if to render my cool page, my controller action tries to request something from the Twitter API (say I grab a list of tweets), the end user has to wait until the Twitter API responds before the application server even returns a response. This can delay page loading by 200-500ms *on average*, with 95th percentile times reaching 20 seconds or more, depending on what your timeouts are set at.

For example, what I can tell from this graph is that Mailchimp (purple spikes in the graph to the right) seems to go down a lot. Wherever I can, I need to make sure that my calls to Mailchimp have an aggressive timeout (something like 5 seconds is reasonable). I may even consider coding up a [Circuit Breaker](#). If my app tries to contact Mailchimp a certain number of times and times out, the circuit breaker will trip and stop any future requests before they've even started.

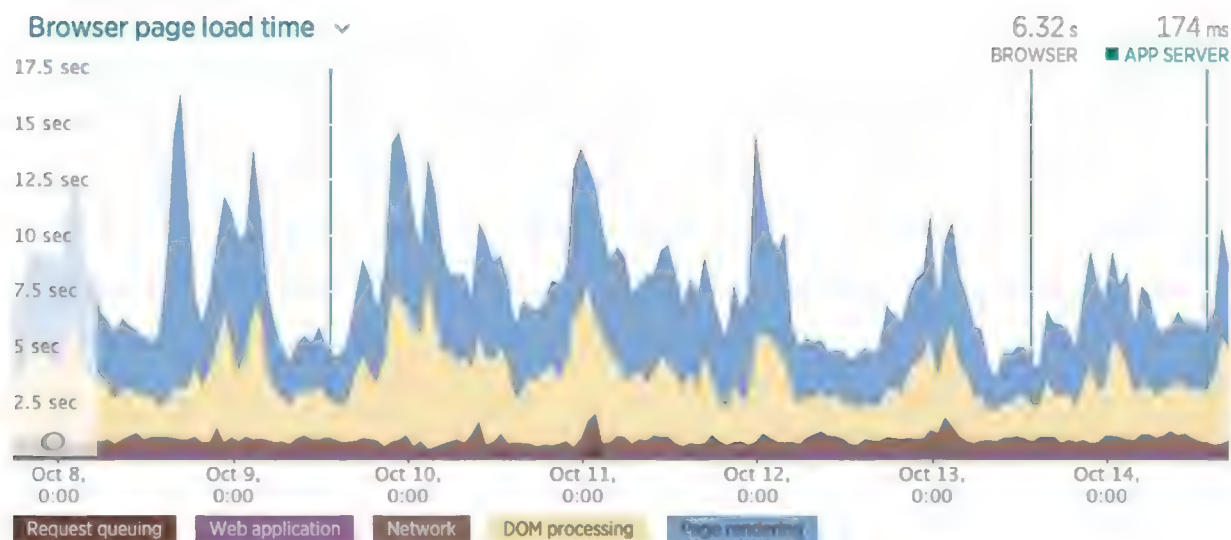
GC stats and Reports

To be honest, I don't find New Relic's statistics here useful. You're better off with a tool like `rack-mini-profiler` and `memory_profiler`. I don't find New Relic's "average memory usage per instance" graph accurate for threaded or multi-process setups either.

If you're having issues with garbage collection, I recommend debugging that in development rather than trying to use New Relic's tools to do it in production. [Here's an excellent article](#) by Heroku's Richard Schneeman about how to debug memory leaks in Ruby applications.

In addition, I'm not going to cover the Reports, as they're part of New Relic's (rather expensive) paid plans - they're pretty self-explanatory.

Browser / Real user monitoring (RUM)



Remember how we applied an 80/20 mindset to the top offenders in the web transactions tab? We want to do the same thing here. Change the timescale on the main graph to the longest available. Instead of the percentile graph (which is the default view), change it to the "Browser page load time" graph that breaks average load time down by its components.

- **Request queueing** Same as the web graph. Notice how little of an impact it usually has on a typical Ruby app - most queueing times are something like 10-20ms, which is just a minuscule part of the average 5 second page load.
- **Web application** This is the entire time taken by your app to process a request. Also notice how little time this takes out of the entire stack required to render a webpage.
- **Network** Latency. For most Ruby applications, average latency will be longer than the amount of time spent queueing and responding! This number includes the latency in both directions - to and from your server.
- **DOM Processing** This is usually the bulk of the time in your graph. DOM Processing in New Relic-land is the time between your client receiving the full response and the `DOMContentLoaded` event firing. Now, this is *just* the client having loaded and parsed the *document*, not the CSS and Javascript. *However*, this event is usually delayed while synchronous Javascript executes. WTF is synchronous Javascript? Pretty much anything without an `async` tag. [For more about getting rid](#)

of that, check out [Google](#). In addition, `DOMContentLoaded` usually also gets slowed down by external CSS. Note that, in most browsers, the page pretty much still looks like a blank white window at this point.

- **Page Rendering** Page Rendering, according to NewRelic, is everything that happens between the `DOMContentLoaded` event and the `load` event. `load` won't fire until every image, script, and iframe is fully ready. The browser *may* have started displaying at least parts of the page before this is finished. Note also that `load` always fires *after* `DOMContentLoaded`, the event that you usually attach most of your Javascript to (Query's `$(document).ready` attaches functions to fire after `DOMContentLoaded`, for example).

For a full guide to optimizing front-end performance issues you find here, see my extensive guide on the topic.

It's important to note that while most users won't see *anything* of your site until at least DOM Processing has finished, they probably will start seeing *parts* of it during Page Rendering. It's impossible to know just how much of it they see. If your site has a ton of images, for example, Page Rendering might take *ages* as it downloads all of the images on the page.

Note also that Turbolinks and single-page Javascript apps pretty much break real-user-monitoring, because all of these events (`DOMContentLoaded`, `DOMContentLoaded`, `load`) will only fire *once*, when the page is initially loaded. New Relic *does* give you additional information on AJAX calls, such as throughput and response time, if you pay for the Pro version of the Browser product.

Conclusion

NewRelic, and other production performance monitoring tools like it, is an invaluable tool for the performance-minded Rubyist. You simply cannot be serious about speed and not have a production profiling solution installed.

As a takeaway, I hope you've learned how to apply an 80/20 mindset to your Ruby application with NewRelic. This mindset can be applied at all levels of the stack, but don't forget - profiling that isn't based on what the end-user experience isn't based in reality. That's why, for a browser-based application, we should be paying attention first to our *browser* experience, not to our backend, even if that's sometimes easier to measure.

Checklist for Your App

- You should be using a performance monitor in production - NewRelic, Skylight, and AppNeta are all respected vendors in this space. It doesn't really matter *which* you use, just use one of them.
- Figure out where your application sits in my performance categories on the front-end and app server - are you below or above average?
- Use NewRelic to look for these performance problems: network calls during a request, your top 5 worst web transactions, and your top time consumers on in the Browser/RUM and app server histograms.

Ruby App Performance Measurement with Skylight

New Relic, as covered in the other lesson, is a great tool. But, it can sometimes feel a little overwhelming. There's a lot in there. And you get the sense it's not designed for Ruby or Rails web applications - it's a one-size-fits-all tool for all kinds of web apps.

Enter Skylight. Created by Tilde.io, a consulting shop with a few employees you may have heard of (Yehuda Katz, former Rails core, Leah Silber, former Bundler/Merb contributor, Carl Lerche, current Rails core, and Godfrey Chan, current Rails core), Skylight.io aims to be a production performance profiler *exclusively* targeted at Rails applications. Interestingly, Skylight does *not* really support bare Rack applications, though it does support [Sinatra](#) and [Grape](#). We'll get into why and how Skylight's instrumentation works later on.

This plucky little upstart seemed interesting enough, and the team behind it looked great, so I decided to take a dive in and give you the Skylight counterpart to the New Relic lesson. We're going to look at how to use Skylight to identify and solve performance issues, and talk about when Skylight might be appropriate for your application.

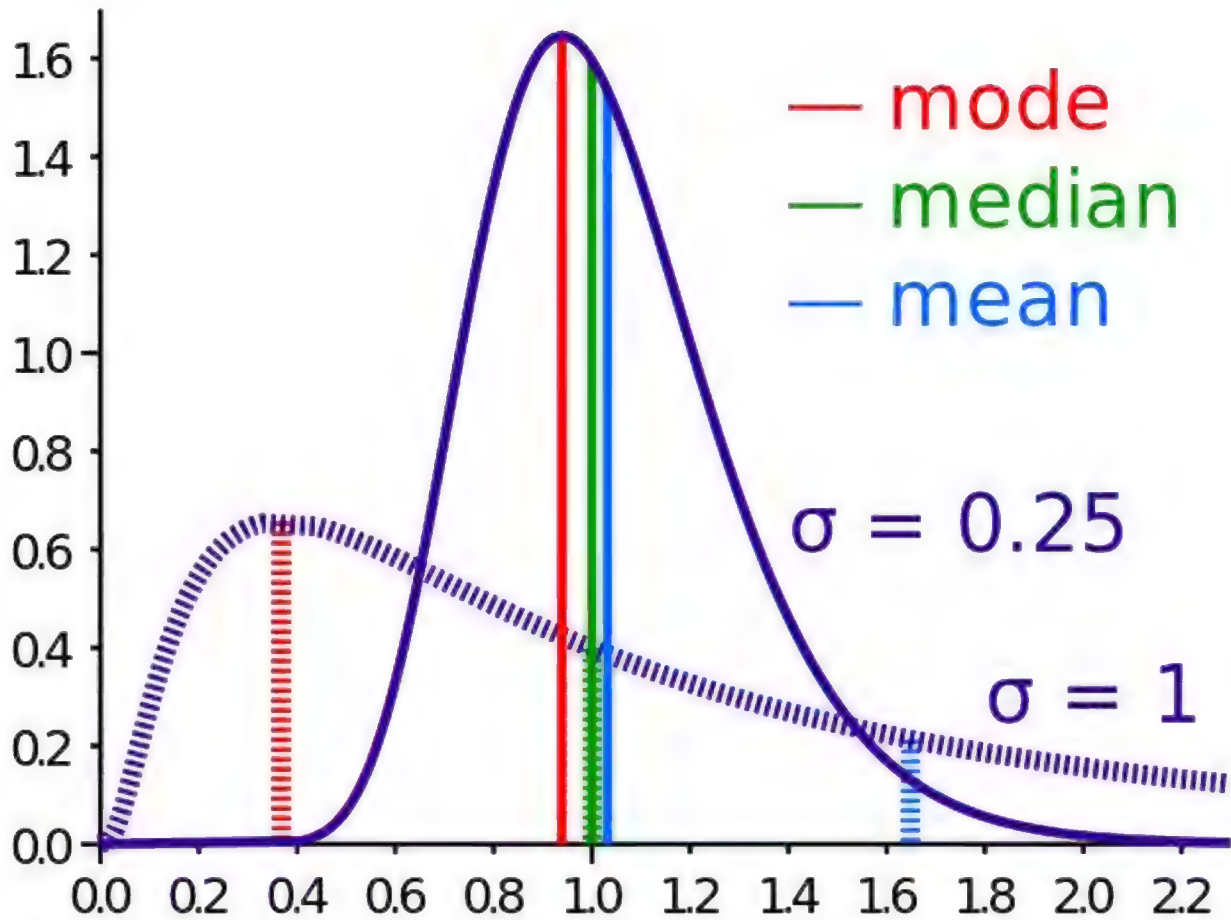
You need to be using a performance monitoring tool in production. Currently, the only real competition I can see is between New Relic and Skylight, at least for Ruby web applications, though there are rumblings about the newcomer [AppNeta](#). Honestly, it doesn't matter which of these tools you use - but you do need *something* recording the experience of production end-users. Flying blind isn't an option.

As with my New Relic lesson, no one from Tilde.io was involved in the creation of this lesson, and I have no relationship with that company.

Skylight's Philosophy - Focus on the Long Tail

Skylight has a slightly unusual philosophy on performance metrics - they don't concentrate on the averages. Skylight is designed - both in their UI and in the design of their profiler - to catch the *worst* of the performance issues happening in the *worst* cases, not in the "average".

Tilde.io team member Godfrey Chan wrote a fascinating post about what he calls "[The Log-Normal Reality](#)". The gist is that while we are taught, since we were children, to think in terms of bell curves and the normal distribution, in reality, most statistical distributions are log-normal. The weird thing here is that the distribution is heavily, heavily skewed. A lot of the things we learned in middle school statistics no longer apply.



Most web applications have log-normal distributions of response times. Here's a response time graph I pulled, at random, from an application I'm working on:



Why do most web applications have log-normal response time distributions?

- **Cached responses** create a peak near the left side of the histogram. When caches are warm, many responses take just a few milliseconds.
- **Network conditions/external services** may fail or go "partially down", leading to bad slowdowns. For example, if you have an action that depends on grabbing something from a Redis server, 99% of that time, it will be fast and work just great.

But 1% of the time, the TCP/IP network demons are going to get you, and that 20ms request will take 300ms. Or perhaps you've got a search endpoint where certain weird combinations of search parameters cause your database to grind to a halt, but the average user is searching simple things like "cat" and "dog".

So far, this probably all conforms with your experiences. So what then, does Skylight do in the face of the "log-normal" reality?

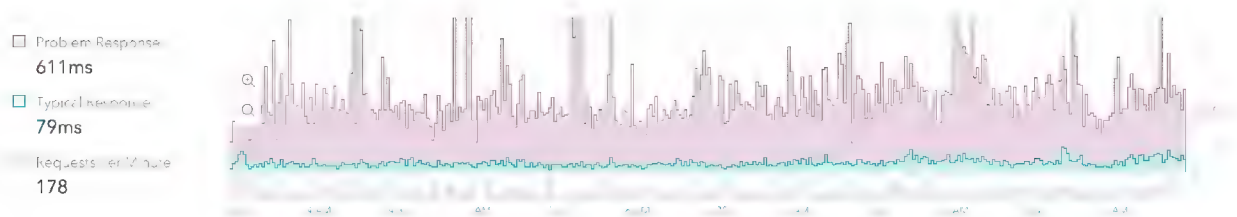
- **Skylight uses logarithmic scales everywhere.** No, really. Nearly everywhere they can.
- **Skylight focuses on 95th percentile times.** The average response time could also be called the "50th percentile" time - 50% of users will get response times faster than this time, and 50% will get response times that are slower. The 95th percentile means that 95 percent of requests are faster than this, but 5% are slower. Skylight's philosophy is that if an average user clicks around your site 20 times, they're probably going to experience at least one of these 95th percentile response times, and so you should focus on these times. [DHH blogged in 2009](#) that averages are useless in most cases. [Here's an interesting Twitter conversation between DHH and Yehuda on the same topic.](#)
- **Instead of sampling, Skylight aggregates.** *Sampling* is when we profile the performance of only a *statistically significant* number of requests. To pull a number out of the air, let's say we only instrument 10% of actual requests. This is fine for tracking averages, but for figuring out what happens in the 95th or 99th percentiles, this isn't enough - we need much higher numbers of samples. Skylight doesn't sample at all - every response is profiled! They can do this because of the design of their agent, which I'll get into later.

There's a couple of terms Skylight uses that are unique to their service. Here's what they mean:

- **Problem Response:** This is the 95th percentile.
- **Typical Response:** The average/median.
- **Popularity:** A logarithmic representation of how often this endpoint appears in Skylight's reports.
- **Agony:** Just a combination of the endpoint's popularity and its response times. Note that Skylight marks endpoint popularity with a *logarithmic scale*, which means that the agony measure is also logarithmic. An endpoint with 2 "exclamation points" of Agony is not twice as bad as a "1 exclamation point" endpoint, it's likely much worse.

Identifying Problems

Skylight's "welcome mat", when you open up an app, is this response time graph:

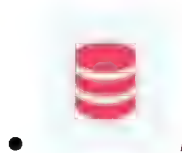


Just like with New Relic, being able to "read" graphs such as this one is a critical performance skill. Be on the lookout for:

- **Typical vs. Problem ratios.** At time of writing, the scale on this graph is still linear and not logarithmic - you're going to want to see the "Problem" response times as close to the average as possible. A ratio of 2-4x should be expected - much worse and you've got long-tail problems.
- **Absolute terms - are we fast or slow?** See the New Relic lesson for some guides as to what you should expect a "fast" or "slow" Ruby app to perform at in terms of milliseconds.
- **Unusual spikiness in "problem" responses.** Both the "normal" and "problem" graphs should be fairly flat. If your app has fairly low load (<100 RPM) and you're seeing graphs that look "spiky", it probably means you have an endpoint that's both slow and infrequently hit. Drill down on the time periods when you see spikes in the graph that are not related to load - most of the time it will be a particular endpoint that's causing the problem.
- **Patterns related to load.** If you have spikes in your graphs at the same time as your RPM increases, you have scale issues. Most likely, you just need more server instances.


Skylight does an *excellent* job of sorting controller endpoints by what they call "agony", as defined above. I find Skylight's "agony" measure extremely accurate as an ordering for my "performance to-do list", where the controller actions with the most "agony" are the places for me to focus my optimizing powers.

Skylight also uses some icons to denote special problems with certain actions:



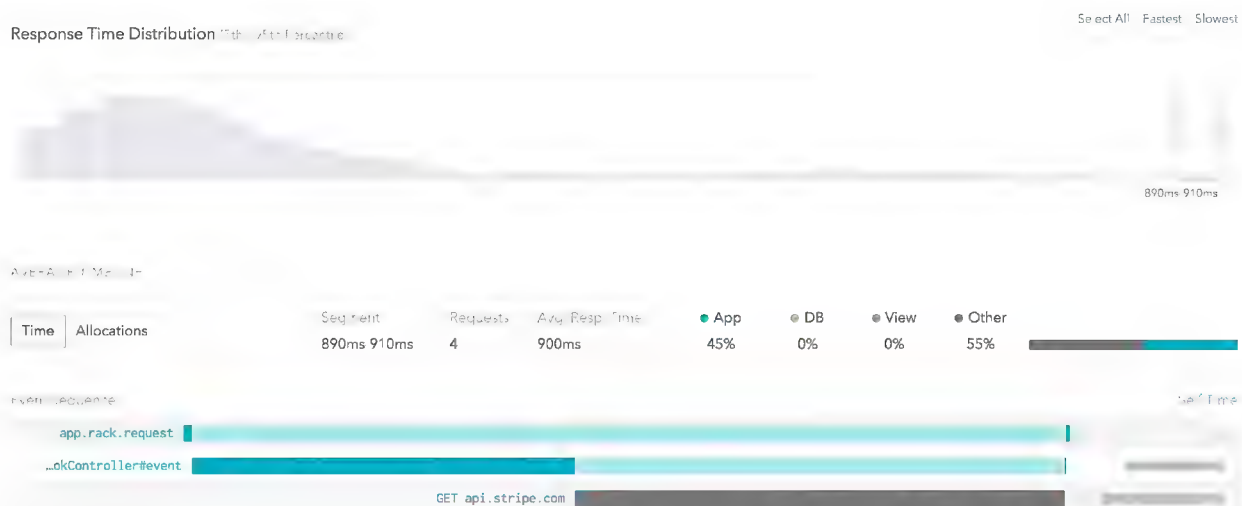
- *Repeated SQL Queries:* This is Skylight's way of saying "I think you have

an N+1 query here!" Fire up your application in development and check these endpoints with production-like data - do you see SQL queries in the logs that look like N+1's?

- 
High Allocations: Skylight's profiler is unique in that it also contains a *memory profiler* which works extremely well in production. For more about memory profilers, be sure to read the lesson . Skylight will pluck out endpoints that have unusually high numbers of allocated objects here. High allocation counts are bad because allocating objects takes time, and in addition will take time *later* as those objects are inevitably garbage collected.

My *least favorite* part of Skylight is that this homepage view is limited to a 6-hour time view. Longer timescales, like 7 days to 30 days, are not available. This is a real bane for low-traffic applications, where 6 hours of time isn't really a great indication of "typical" traffic to your site.

Click an endpoint to see the trace for that individual endpoint. Be sure to click-and-drag on the histogram, which will modify the trace to show you traces for *only that section of the histogram*. This is probably the *best* tool available right now for tracking down exactly what happens in 95th percentile web requests. It's probably one of my favorite parts of Skylight (besides the Agony sort).



There's also a button here for showing the trace with *object allocations* as the fundamental unit, rather than time. Although I haven't personally been able to use this tool yet, it seems like it could be extremely effective for tracking down bloated controller endpoints that allocate unusually large numbers of objects.

Under the hood, [Skylight uses ActiveSupport::Notifications](#) to track what's going on in a Rails application. This is an interesting approach, because *anything* in your application and its libraries can emit ActiveSupport::Notifications and Skylight will pick up on them. Otherwise, you can write your own custom "probes".

How Does it Compare?

The bit you've been waiting for - is it better than New Relic? After using Skylight, I have to say that's a bit of an apples-and-oranges comparison.

First, the length of this lesson should tell you a lot. There's not all that much to Skylight - really, it's mostly just the "web transactions" feature of New Relic *as an entire product*. Tilde.io thinks this is a good thing, and I would agree to an extent - the web transactions view is my #1 most visited screen in New Relic.

Ultimately, you'll have to decide for yourself whether Skylight's feature list cuts it for you. Here are some things I use often in New Relic that aren't present (as of writing, January 2016) in Skylight:

- Breakdown of average response times by component - time spent in GC, request queue, Ruby, database and cache store.
- Apdex scores, which I find to be a fairly useful measure of standard deviation and indicator of "partial downtime".
- New Relic's database view is extremely powerful for tracking down particular queries that are slow app-wide.
- New Relic's external services view is great for seeing which external APIs that you depend on are slow.

However, there are some areas where I think Skylight is unquestionably better:

- Skylight's agent (the process that runs alongside your Ruby process and takes measurements) is considerably lighter-weight than NewRelic's Ruby agent. Skylight's agent is written in Rust, which means it's considerably faster and takes up less memory than NewRelic's agent. I'd estimate 10-30MB of memory savings.
- Because the agent is so lightweight, it allows Skylight to measure *every request*, rather than New Relic's agent, which has to *sample* certain kinds of statistics for your dashboard. New Relic samples their transactions traces and SQL traces - these are running on *every request* with skylight. I far prefer Skylight's aggregation approach - it just makes tracking down 95th/99th percentile issues so much easier.
- I like Skylight's "opinionated" view of your data. The "sort by agony" feature has

been extremely accurate and useful for me. Your mileage may vary.

Skylight's pricing is request-based - up to 100,000 requests per month is free. That's roughly 2 requests-per-minute. From there, it scales up. A 200 RPM app would be paying about \$250 a month. That's not cheap, in my opinion, especially when I can get the same monitoring for free, forever, with an *unlimited* amount of dynos, from NewRelic.

To sum up - while expensive, Skylight's ease of use and the quality and presentation of its data make it a worthy competitor to NewRelic. I suspect that the choice of one or the other will be dependent on the application and problem domain - only you will know for sure what your application needs.

Checklist for Your App

- You should be using a performance monitor in production - NewRelic, Skylight, and AppNeta are all respected vendors in this space. It doesn't really matter *which* you use, just use one of them.
- Try Skylight - paying particular attention to their unique allocation tracing. The agent is lightweight enough to run alongside NewRelic without problems.

Module 2: Front-End Optimization

This module is about the principles of front-end performance - what happens in the browser. This area is probably the most overlooked by full-stack developers. Frequently, we just expect "the front-end people" at our company to deal with this problem. A great number of us don't have dedicated "front-end people".

When it comes to end-user experience, nothing matter more than front-end performance. Back-end performance is often just a tiny component of a user's overall perceived load time. Server responses on a Rails application are often in the ballpark of 100-200 milliseconds. Add in 100 milliseconds of network latency, and you get, generously, about 300 milliseconds. However, front-end load times are often 2 to 5 seconds - making backend performance just 10 percent or less of that total! I urge you to pay close attention to this module for these reasons.

The most important lesson in this module is on Chrome Timeline - understanding how to measure and profile the performance of your front-end is far more important than understanding a single technique or trick. If you deeply understand how to test and experiment with the performance of your site, you'll be able to implement any of the strategies in the remainder of this module.

Chrome Timeline, Your Front-end Profiler

Server response times, while easy to track and instrument, are ultimately a meaningless performance metric from an end-user perspective. End-users don't care how fast your super-turbocharged bare-metal Node.js server is - they care about the page being completely loaded as fast as possible. Your boss is breathing down your neck about the site being slow - but your Elixir-based microservices architecture has average server response times of 10 nanoseconds! What's going on?

Well, what does constructing a webpage actually require? The server has to respond with the HTML (along with the network latency involved in the round-trip), the JS, CSS and HTML needs to be parsed, rendered, and painted, and all the Javascript tied to the page ready event needs to be executed. That's actually a lot of stuff. Usually, server response times make up only a small fraction of this total end-user experience, sometimes as little as 10%. In addition, it's easy for any of these steps to get out of hand quickly:

- Server response times can easily balloon without proper use of caching, both at the application and HTTP layers. Bad SQL queries in certain parts of the application can send times skyrocketing.
- JS and CSS assets must be concatenated, minified and placed in the right place in the document, or rendering may be blocked while the browser stops to load external resources (more on this later). In addition, these days when there's a JQuery plugin or CSS mixin for just about anything, most developers have completely lost track of just how much CSS and JS is being loaded on each page. Even if, gzipped and minified, your CSS and JS assets are <100kb, once they're un-gzipped, they *still* must be parsed and loaded to create the DOM and CSSOM (explained in more detail below). **While gzipped size is important when considering how long CSS or JS will take to come across the network, *uncompressed* size is important for figuring out how long it will take the client to parse these resources and construct the page.**
- Web developers (especially non-JavaScripters, like Rails devs) have an awful habit of placing tons of code into `$(document).ready();` or otherwise tying Javascript to page load. This ends up causing *heaps* of unnecessary Javascript to be executed on every page, further delaying page loads.

So what's a good, performance-minded full stack developer to do? How can we take our page loads from slow to ludicrous speed?

But, rather than just *tell you* that XYZ technique is faster than another, I'm going to *show you* how and *why*. Rather than take my word for it, you can test different front-end optimizations for yourself. To do that, we're going to need a profiling tool.

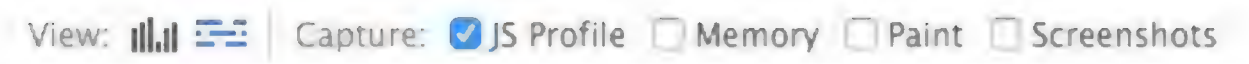
Enter Chrome Timeline

My number one front-end performance tool is [Chrome Timeline](#). While I use New Relic's real user monitoring (RUM) to get a general idea of how my end-users are experiencing page load times, Chrome Timeline gives you a millisecond-by-millisecond breakdown of exactly what happens during any given web interaction. Although I'm going to show you how to use Chrome Timeline to analyze page loads, you can also use it to profile Javascript interactions once the page has loaded.

Note that most of Google's documentation on Chrome Timeline is severely out of date and shows a "waterfall" view that no longer exists in Chrome as of October 2015 (Chrome 45). This post is up-to-date as of that time.

Chrome Timeline *also* works really well for optimizing "60fps" JavaScript applications. I'm not going to get into that here. What I'm going to discuss is how we can use Chrome Timeline to make our applications take as little time as possible between user input (clicking, pushing a button, hitting enter) and response (displaying data, moving us to a new page, etc), focusing on the initial page load.

To open Chrome Timeline, open up Chrome Developer Tools (Cmd + Alt + I on Mac) and click on the Timeline tab. You'll see a blank timeline with millisecond markings. For now, uncheck the "causes", "paint" and "memory" checkboxes on the top, and disable the FPS counter by clicking the bar graph icon, like this:



These tools are mostly useful for people profiling client-side JS apps, which I won't get into here.

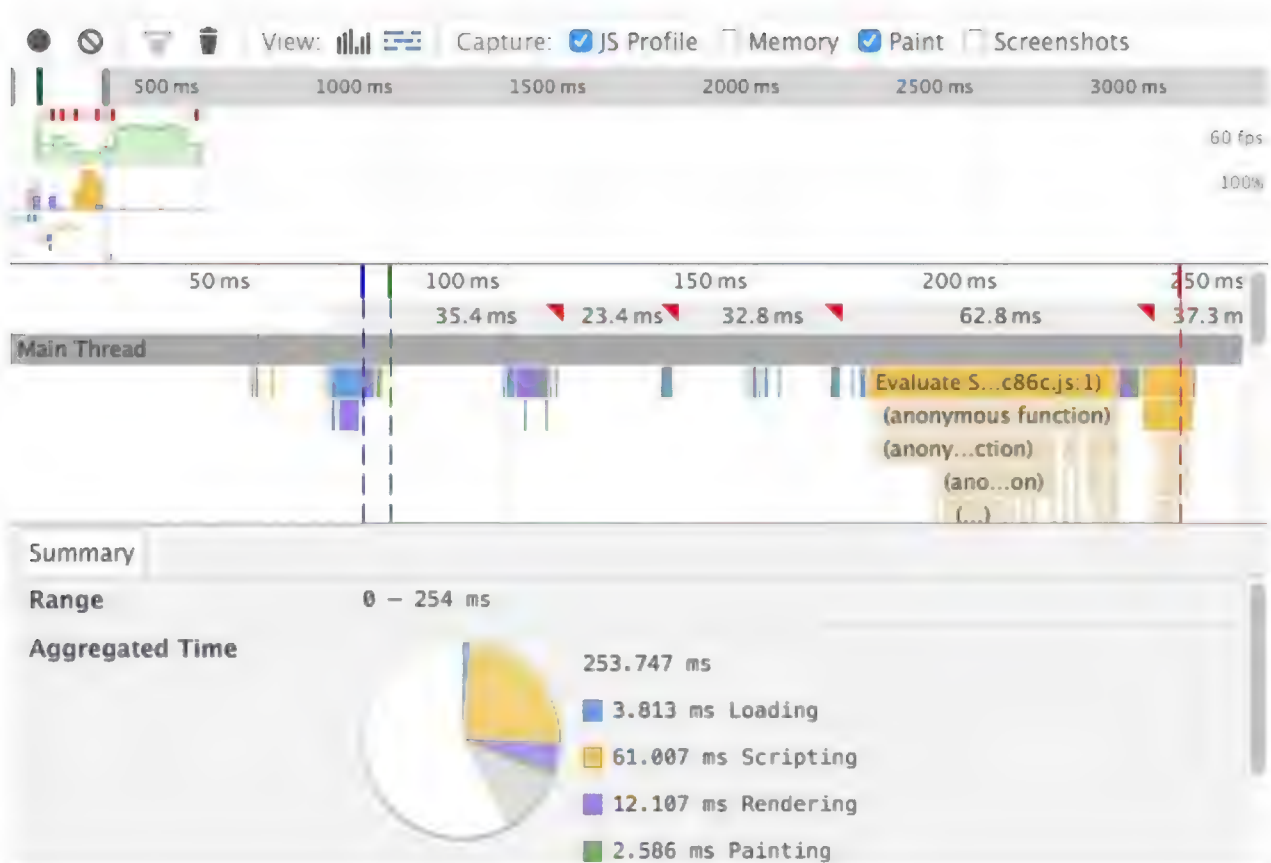
The Chrome Timeline records page interactions a lot like a VCR. You can click the little circular icon (the record button) at any time to turn on Timeline recording, and then click it again to stop recording. If the Timeline is open during a refresh, it will automatically record until the page has loaded.

Let's try it on <http://todomvc-turbolinks.herokuapp.com/>. This is a [TodoMVC](#) implementation I did for a previous blog on Turbolinks. While the Timeline is open, you can trigger a full page load with CMD + Shift + R and Chrome will automatically record the page load for you in Timeline. Be sure you're doing a hard refresh here, otherwise you may not redownload any assets.

Note that browser extensions will show up on Chrome Timeline. Any extension that alters the page may show up and make your timelines confusing. Do yourself a favor and disable all of your extensions while profiling with Chrome Timeline.

We're going to start with a walkthrough of a typical HTML page load in Timeline, and then we're going to identify what this performance profile says about our application and how we can speed it up.

Here's what my Timeline looked like:



254 ms from refresh to done - not bad for an old Rails app, eh?

Receiving the HTML

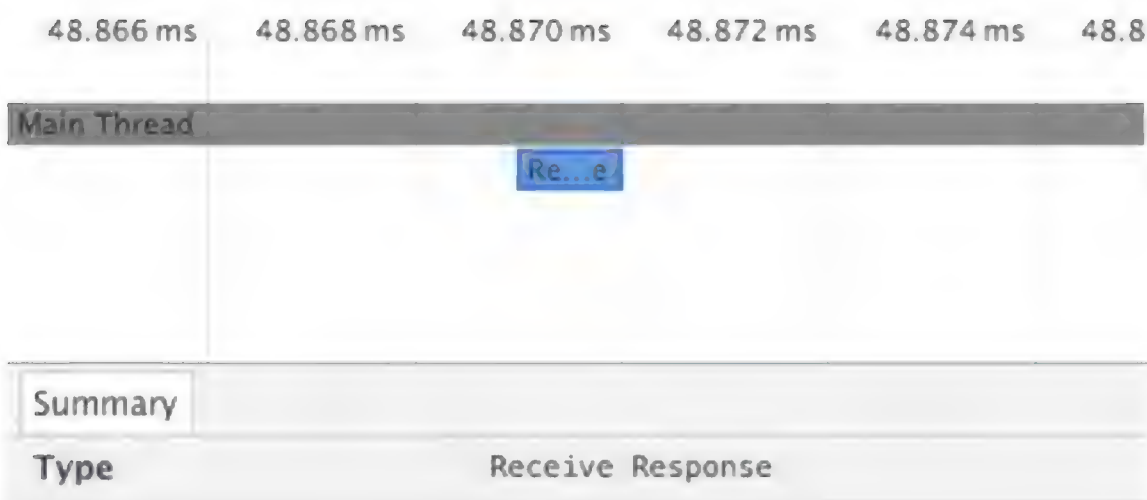
The first thing you'll notice is that big chunk of idle time at the beginning. Almost nothing is happening until about 67ms after I hard-refreshed. What's going on there? It's a combination of server response time (on this particular app, I know it hovers around 20ms), and network latency (depending on how far you are from the US East Coast, anywhere from 10-300ms).

Even though we live in an age of mass cable and fiber optic internet, our HTTP requests still take a lot of time to go from place to place. Even at the theoretical maximum speed of an HTTP request (the speed of light), it would take a user in Singapore about 70ms to reach a server in the US. And HTTP doesn't travel at the speed of light - cable internet works about half that speed. In addition, they make as many as a dozen intermediate stops along the way along the Internet backbone. You can see these stops using `tracert`. In addition, you can get the approximate network latency to a given server by simply using `ping` (that's what it was designed for!).

For example, I live in New York City. Pinging a NIST time server in Oregon, I usually can see network latency times of about 100ms. That's a pretty substantial increase over the time we'd expect if the packets were traveling at the speed of light (~26ms). By comparison, my average network latency for a time server in Pennsylvania is just 20ms. And Indonesia? Packets take a whopping 364ms to make the round trip. For websites that are trying to keep page load times under 1 second, this highlights the importance of geographically distributed CDNs and mirrors.

Let's zoom in on the first event on the timeline. It seems to happen in the middle of this big idle period. You can use the mouse wheel to zoom.

The first event on the Timeline is "Receive Response".



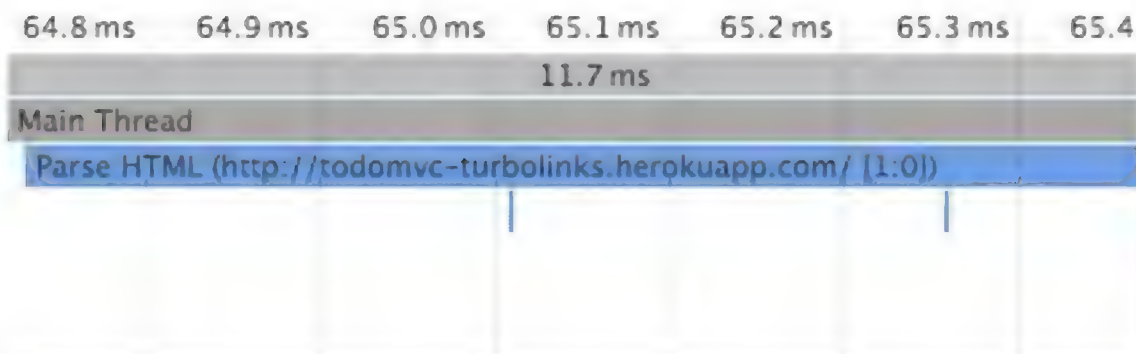
A few milliseconds later, you'll see a (tiny) "Receive Data" event. You might see one or two more miscellaneous events related to page unloading, another "Receive Data" event, and finally a "Finish Loading" event. What's going on here?

The server has started responding to your request when you see that first "Receive Response" event. You'll see several "Receive Data" events as bytes come down over the wire, completing with the "Finish Loading" event. This pattern of events will occur for any resource the page needs - images, CSS, JS, whatever. Once we've finished downloading the document, we can move on to parsing it.

Parse HTML

"Parsing HTML" sounds like a pretty simple process, but Chrome (and any browser) actually has a lot of work to do. The browser will read the bytes of HTML off the network (or disk, if you're viewing a page on your computer), and convert those bytes into UTF-8 or whatever document encoding you've specified. Then, the browser has to "tokenize" - basically taking the long text string of the HTML and picking out each tag, like `` and `<a>`. Imagine that the browser converts the ~100kb string of HTML into an array of several strings. Then it "lexes" these tokens (basically converts them into fancy objects) and finally constructs a DOM out of them. On complicated pages, these steps add up - on my machine, The Verge takes over 200ms just to *parse the HTML*. Yow.

You may also see two "Send Request" events (they're really small) beneath the "Parse HTML" event. In case you haven't figured it out already, what we're looking at is called a "flamegraph". Events underneath other ones mean that the upper event "called" the lower one. The two "Send Request" events you see here are the browser requesting the Javascript and CSS files linked in the head. This is a Rails app, so there's only one of each.



In addition, the Javascript file in this app is marked with an `async` attribute:


```
<script src="/assets/application.js" async="async" data-turbolinks-track="true"></script>
```

Normally, when a browser sees a Javascript tag like this in the head, it *stops completely* until it has finished downloading and evaluating the script. If the script is remote, we have to wait while the script downloads. This can take *a lot* of time - even more than a whole second, when you include network latency and the time required to evaluate the script. The reason browsers do this is because Javascript can modify the DOM - any time there's a script tag, the browser has to execute it because it could change the DOM or layout. For more about Javascript blocking page rendering, [Google does a great explanation here](#).

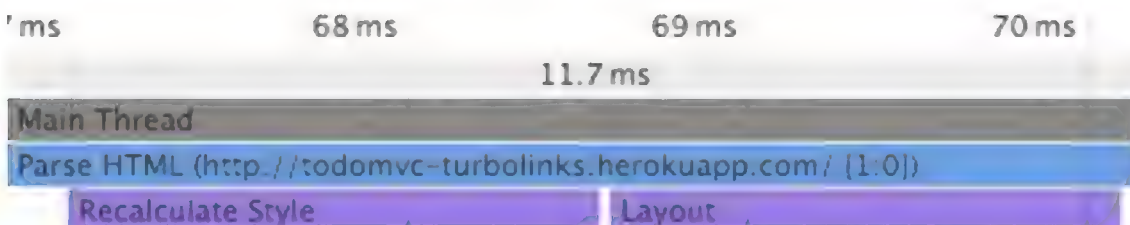
Because this script tag was marked with the `async` attribute, this doesn't happen - the browser won't "stop the world" to download and evaluate the Javascript. This can be a *huge* boost to speeding up time-to-first-paint for most websites.

Browsers will *not* wait on external CSS before continuing past this step. If you think about it, this makes sense. CSS cannot modify the DOM, it can only style it and make it pretty. In order to even apply the CSS, we need to have the DOM constructed first. So the browser, smartly, simply sends the request for the CSS and moves on to the next step.

Note that this "Parse HTML" step will reoccur every time the browser has to read new HTML - for example, from an AJAX request.

Recalculate Styles

The next major event you're going to see is the purple "Recalculate Styles". Unfortunately, this event covers a lot of things that actually happen during page construction. The first is the construction of the CSSOM.



As HTML is to the DOM, so CSS is to the CSSOM. Your CSS, after it's downloaded has to be converted -> tokenized -> lexed -> constructed just like the HTML was. This process is usually the cause of any "Recalculate Styles" bars you see at the beginning of the page load.

"Recalculate Styles" can also mean a lot of other confusing things are happening with your CSS, like "recursive calculation of computed styles", or whatever that means. The gist is that if you're seeing a lot of time in "Recalculate Styles", your CSS is too complicated. Try to eliminate unused or unnecessary style rules.

Why are we seeing Recalculate Styles events when the CSS hasn't even been downloaded yet? The browser is applying the browser's default CSS to the document, and it may also be applying any `style` attributes present in the HTML markup itself (`display: none` being a common one, present on this page).

You will probably see more purple events (Recalculate Styles and its cousin, Layout) later on in the timeline. Again, your browser does not wait for CSS to finish downloading - it's already calculating styles and layouts based on just your HTML markup and the browser defaults right now. The rendering events you see later on occur once the CSS is finished downloading.

Layout

Slightly after your first Recalculate Styles event, you should see a purple "Layout" event. Basically, at this point, your browser has all of the DOM and CSSOM in memory and needs to turn it into pixels on the screen.

The browser traverses the visible elements of the DOM (actually the render tree), and figures out each node's visibility, applicable CSS styles, and relative geometry (50% width of its parent and so on). Complicated CSS will make this step longer, but so will complicated HTML.

If you're seeing a lot of "layout" events during a page load, you may be experiencing something called **"layout thrashing"**. Any time you change the geometry of an element (its height, width, whatever), you trigger a layout event. And, unfortunately, browsers can't tell what part of the page they need to recalculate. Usually, they have to recalculate the layout for *the entire document*. This is especially slow with float-based layouts, though it's slightly faster with flex box layouts. Layout thrashing is usually going to be caused by Javascript messing with the DOM, though using multiple stylesheets will also cause it. [For more about layout thrashing, Google has an excellent page on the topic.](#)

In summary - in the "Layout" step, then, the browser is just calculating what's visible, what isn't, and where it should go on the page.

DOMContentLoaded

It's generally at this point that you'll see the blue bar in Timeline - this is the `DOMContentLoaded` event. At this point, your browser is done parsing the HTML and running any blocking Javascript (that is, Javascript either embedded in the page or in a script tag that isn't marked `async`). Most browsers have not painted *anything* to the screen by this point.

To speed up `DOMContentLoaded`, you can do a few things:

- Make script tags `async` where possible. Moving script tags to the end of the document doesn't help speed up `DOMContentLoaded`, as the browser must still evaluate the Javascript before completing the construction of the DOM. All "async" means is that the only part of the script executed "synchronously" is the start of downloading of the script itself, its execution will be delayed until later. [Ilya Grigorik suggests that using `async` tags is generally cleaner and more effective than using so-called 'async' script injection.](#)
- Use less complex HTML markup.
- Avoid layout thrash (see above). Don't use more than one stylesheet - concatenate your assets!
- Inline styles in moderation. Inlining styles means that the browser may try to parse the stylesheet before moving on to the rest of the document. Google recommends inlining only styles required to display above-the-fold content. This will slow down `DOMContentLoaded` but will speed up the window's `load` event. This may be true, but you certainly don't want to inline *all* of your CSS. Also, figuring out what CSS rules you need for the above-the-fold content in this age of CSS frameworks and Bootstrap sounds like a *lot* of work to me. How much CSS do you need to render above-the-fold? *All of it*. As a rule of thumb, don't consider inlining *all* of your CSS unless you've got about 50kb or less of it. Once HTTP2 becomes more common and we can download CSS, HTML and JS over the same connection, this optimization will no longer be needed.

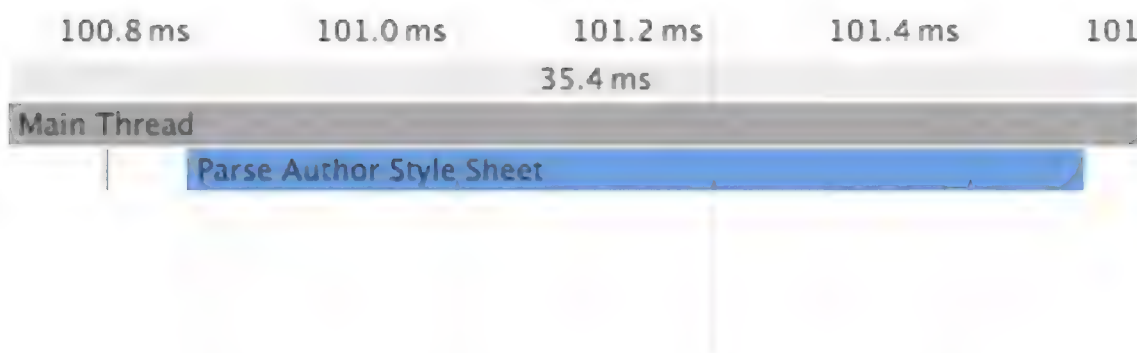
Paint

As we move along the timeline to the right, you should start seeing some green bars in the flamegraph. These are Paint related events. There's a *whole* lot that can go on in these events (and Chrome even provides profiling tools just for these painting events), but I'm not going to go too deep on them here. All you need to know is that paint events happen when the browser is done rendering (the purple bars - the process of turning your CSS and HTML into a layout) and needs to turn the layout into pixels on a screen.

The green bar in the timeline is the first paint - the first time anything is rendered to screen. Optimizing first paint is largely a matter of optimizing DOMContentLoaded and getting the stylesheet to the client as fast as possible. Any stylesheet that doesn't specify a media query (like `print`) will block page rendering until we've downloaded it and parsed it.

Parse Author Style Sheet

Keep scrolling to the right on the Timeline. Wow - see how much longer it took to get to this part?



In my case, it took almost 40 ms of just waiting around to download the whole stylesheet - and this app's stylesheet isn't even that big! To be exact, we sent the request for the stylesheet at about 65ms, and it didn't come back until 101ms. In reality, this actually extremely fast (in a real app, you would expect that to be more like 200-350ms at least), and we can't really optimize that much further. I'm in NYC and Heroku is in Virginia, so most of that time is network latency anyway.

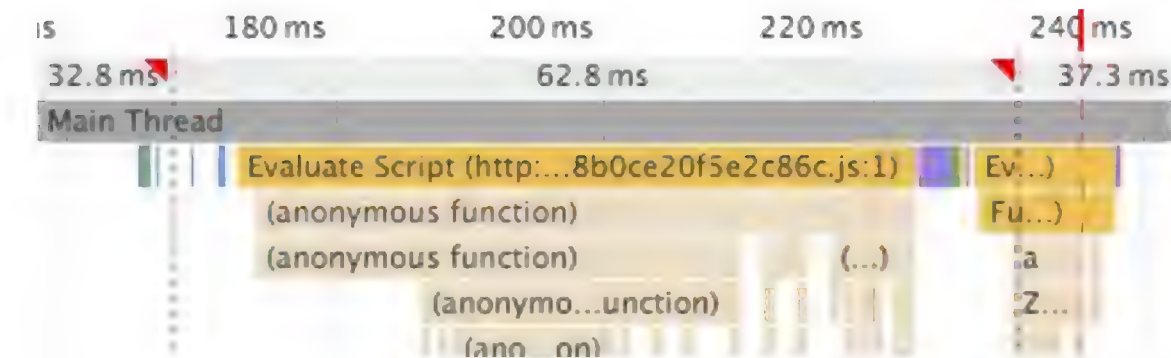
Once the stylesheet is downloaded, it's parsed. You'll see another cycle of purple events (as the CSSOM is re-calculated, we re-render the layout) and green events (now that the layout is updated, we render the result to the screen).

The stylesheet for this app is extremely simple, and my app appears to be wasting about 30ms waiting for the CSS to download. It may be worth investigating the performance impact of inlining the entire stylesheet in the HEAD of this page. Most sites won't benefit

from this optimization (see my bit about this above), but because this app is idling for about 20ms waiting for the styles to download, we may want to eliminate that network round-trip.

Javascript

Eventually, you'll notice the Javascript finish downloading (this is the "Finish Loading" event for your Javascript file).



A millisecond or two after this occurs, you'll see the big yellow "Evaluate Script" bars start up. You'll notice the flamegraph start to get a lot deeper here. It's hard to tell on this site as to what's going on because the Javascript has been minified, but in development mode, pre-minified, you can learn a lot about why it takes so long for your Javascript to evaluate here.

Note that this is a really, really simple application, but because of the sheer amount of Javascript involved, it takes 76ms for my machine just to parse and evaluate it all. Remember that this will happen on *every page load*, and *double* the amount of time on a mobile browser. This isn't even that much JavaScript in web terms - 37kb gzipped.

Eventually, after a whole lot of script evaluation, you'll probably see a couple of Recalculate Style and Paint events. Your Javascript will probably do a few things to change the layout - that's what's happening here.

Finally, you should see the `load` event fire off. There will be several Javascript functions attached to this event in almost every application.

Once all of those callbacks attached to `load` have completed, you'll see the **red bar**, which signifies the end of the `load`. This is generally when the page is "ready" and finished loading. Finally!

Using Chrome Timeline to Debug Browser Speed

You've got a site that takes 5-10 seconds to get to the `load` event. How can you use Timeline to profile it and find the performance hotspots?

1. **Hard reload (ctrl-shift-r) and load the Timeline with fresh data**
2. **Look at the pie graph for the entire page load.** After hard reloading, Chrome will show the aggregate stats for the entire page load in the pie graph. You can see [here](#) that it took about 2.23 seconds from my refresh input to get to `load`. Get an idea of where you spend most of your time - is it in parsing (loading), scripting or rendering and painting? Is it idle time?
3. **Reduce Idle** Idling comes from slow server responses and asset requests. If you're idling a lot, make sure your server is still zippy-quick. If it is, you may have an unoptimized order of assets. See the "DomContentLoaded" section above.
4. **Reduce Loading** Recall that "loading" here refers to time spent parsing HTML and CSS. To decrease loading time, you don't have many options other than to decrease the amount of HTML and CSS you're sending to the client.
5. **Reduce Scripting** Time spent evaluating scripts is usually the largest chunk of page load time outside of waiting for the network. Most sites use quite a few different marketing-related JavaScript plugins, like Olark and Mixpanel. Where possible, I would try to add `async` tags to these scripts to get them off the rendering critical path, even if the vendor proudly claims the script is already "async!". Try to look at the call stacks and figure out where you're spending most of your time.
6. **Reduce Rendering and Painting** Sites can also have quite a few layout changes and re-renders due to tools like Optimize.ly, something we can see by checking the "First Layout Invalidation" property of some of the "Layout" events in the Timeline. This is a tough one. Optimize.ly's whole purpose is to essentially change the content of the page, so moving it to an `async` script tag may cause a "flash of unstyled content" where part of the page would look one way and then suddenly flash into a different styling. That isn't acceptable, so we're stuck with Optimize.ly's slow and painful re-layouts [here](#).

Checklist for Your App

- **You should have only one remote JS file and one remote CSS file.** If you're using Rails, this is already done for you. Remember that every little marketing tool -

Olark, Optimize.ly, etc etc - will try to inject scripts and stylesheets into the page, slowing it down. Remember that the cost of these tools is not free. However, there's no excuse for serving multiple CSS or JS files from your own domain. Having just one JS file and one CSS file eliminates network roundtrips - a major gain for users in high-latency network environments (international and mobile come to mind). In addition, multiple stylesheets cause layout thrashing.

- Every script tag should have `async` and `defer` attributes. Do not script inject.** "Async" javascripts that download and inject their own scripts (like [Mixpanel's "async" script here](#)) are not truly "asynchronous". Using the `async` attribute on script tags will *always* yield a performance benefit. Note that the attribute has no effect on inline Javascript tags (tags without a `src` attribute), so you may need to drop things like Mixpanel's script into a remote file you host yourself (in Rails, you might put it into `application.js` for example) and then make sure that remote script has an `async` attribute. Using `async` on external scripts takes them off the blocking render path, so the page will render without waiting for these scripts to finish evaluating.
- CSS goes before JavaScript.** If you *absolutely must* put external JS on your page and you can't use an `async` tag, external CSS must go first. External CSS doesn't block further processing of the page, unlike external JS. We want to send off all of our requests *before* we wait on remote JS to load.
- Minimize Javascript usage where possible.** I don't care how small your JS is gzipped - any additional JS you add takes additional time for the browser to evaluate on *every page load*. While a browser may only need to *download* JavaScripts once, and can use a cached copy thereafter, it will need to *evaluate* all of that JavaScript on *every page load*. Don't believe me that this can slow your page down? Check out [The Verge](#) and look at how much time their pages spend executing JavaScript. Yowch.
- `$(document).ready` is not free - eliminate event handlers where possible or use a solution that re-uses the page, like Turbolinks or a single-page-app approach.** Every time you're adding something to the document's being ready, you're adding script execution that delays the completion of page loads. Look at the Chrome Timeline's flamegraph when your `load` event fires - if it's long and deep, you need to investigate how you can tie fewer events to the document being ready. Can you attach your handlers to `DOMContentLoaded` instead?

Lab: Chrome Timeline

This lab requires some extra files. To follow along, download the source code for the course and navigate to this lesson. The source code is available on GitHub (you received an invitation) or on Gumroad (in the ZIP archive).

Provided is the actual homepage for the Ruby language. In this directory, in your shell, type:

```
$ ./serve_lab.sh
```

This will start a local webserver. The `ruby-lang.org` homepage is now available at `localhost:8000`.

Exercise 1

Open `localhost:8000` in Chrome Developer tools. Click the "Network" tab and where it says "No throttling", click and add a new custom throttling preset - 100ms of latency and a 1.5 Mb/s download speed. These numbers represent an average US connection, based on numbers from Akamai's State of the Internet report. If we didn't throttle, the requests would complete unrealistically fast.

Now that throttling is set up, click the Timeline tab and perform a full page refresh (CMD-SHIFT-R on Mac, or right-click the refresh button and select "Hard Reload").

Using the skills gained in this lesson, reduce this page's `DOMContentLoaded` time by 50%. On my machine, `DOMContentLoaded` took 1.13 seconds, and `Load` took 1.58 seconds. If your times are considerably faster, make sure your network throttling is correctly set (there should be a yellow exclamation point next to the network tab).

If you're struggling, try using the Network tab as well.

Run `serve_solution.sh` and compare against your work. Check out the `lab/solution` directory to see what was changed.

The Optimal Head Tag

Most of us developers settle for page load times somewhere between 3 and 7 seconds. We open up the graph in NewRelic or webpagetest.org, sigh, and then go back to implementing that new feature that the marketing people *absolutely must have deployed yesterday*.

Little do we realize, perceived front-end load times closer to half a second are possible for most (if not all) websites with little effort.

Most webpages have slow front-end load times not because they're heavy (north of 1MB), or because they need 200kb of Javascript just to render a "Hello World!" (*cough Ember cough*). It isn't because the pipes are too small either - bandwidth is really more than sufficient for the Web today.

HTML, TCP and latency are the problems, not bandwidth. Page weight, while important, is a false idol.

A 1MB webpage, with all of its scripts and CSS inlined, will load faster than 1 MB webpage with 100 different asset requests spread across 10 domains. Each of these asset requests requires a TCP connection, and setting up those connections takes longer when there's more network latency. This is really TCP's fault - it was designed for long, streaming downloads, not the machine-gun fire of 3rd-party Javascript and assets that most websites today require. God forbid you're in a high-latency environment too, like a mobile connection or a developing country. When latency starts to shoot north of 100 milliseconds, webpages grind to a halt trying to set up dozens of [three-way handshakes](#) to download all of the cat gifs your social media intern said would *totally blow up* this blog post on Reddit.

In addition, some quirks in how HTML works means that certain sub-resources *must* block page rendering - leaving the browser idling, waiting for things to download and execute. Preventing (and dealing with) the various types of blocking that can happen during a webpage load presents a major performance opportunity. The problem of webpage loading is generally not a problem of resources, it's a problem of using those resources efficiently so that they don't block each other's execution.

Humans are squishy, and perceived load times are not the same as window load times. We can hack our user's perceptions to make them *think* the webpage loaded faster than it did. `window.load`, while a good starter metric for measuring page load

speed, is not a realistic interpretation of how users look at webpages. Humans (unlike computers) can begin to understand the webpage before it's even finished completely loading. This means that *time to paint*, not *time to load* is important. In addition, *time to paint the page's usable content* is of course the most important thing. Gmail quickly paints a loading bar, sure, but you didn't come to Gmail to see the loading bar. You came to see the application. Likewise, if our news website paints some divs to the page but doesn't actually show any text until 2 seconds later because the web fonts took forever to load, then the site wasn't really usable until that text was painted. It's easier to decrease *perceived* load times than it is to decrease *total* load time (as measured by `window.load`). Amazon, for example, paints a nearly complete page just 1.5 seconds after a request is sent, but `window.load` doesn't fire until 3.5 seconds later.



We can leverage human perception to disproportionately affect perceived load times with minimal effort. And the place these opportunities can be exploited is in a site's head tag.

The head tag is probably the most important part of any webpage from a performance standpoint. It can truly make or break a speedy page - two identical head tags with different element ordering can have speed differences on an order of magnitude, especially in poor network conditions (like mobile or the developing world). But sometimes optimizing head tags can be confusing - there's a lot to understand and browser technology changes rapidly, meaning yesterday's advice can be out of date.

In this article, I'll attempt to show what the optimal head tag looks like - what elements it contains, in what order, and with what special attributes (such as `async` and `defer`) that will lead to zippy-quick load times.

First, some definitions. What *exactly* are we going to optimize for?

When thinking about page load optimization, there are usually three important times for the end user:

- **First paint** - When does the page first start painting to the screen? This doesn't have to be *all* the content - frequently it looks like just a few colored `div` blocks with no text in them (waiting for the fonts to load). Images are usually not loaded yet. Heck, we may not even have downloaded the CSS for anything below the fold yet (the initial viewport - more on that later). But this time is still important - it's when a user first sees a reaction to their input. Decreasing time-to-first-paint can be a critical optimization in improving user *perception* of page loads. This is why [Facebook hacks the JPEG algorithm to send a blurred, 200 byte version of cover photos on mobile](#). Creating a *perception* of the page loading is just as important as the page *actually* loading.
- **First paint of text content** - Webpages are text-delivery mechanisms. The Web is typography. When does the page start painting text to the screen? As soon as a page's critical text has been painted - before the images have been downloaded or even any decorative elements rendered - the user can begin processing the information on the screen. And not all text content is equal here - painting "Loading..." to the screen doesn't count. A user cannot begin to *do what they came to your website to do* until the text on that page has painted to the screen, making the moment that text appears one of the most important of your website's loading process. This time can often be substantially different than time to first paint, for reasons I'll get into later on. This is a pet theory of mine, and I am not a designer or information architect by trade, so take this all with a grain of salt.
- **The load event** - The `load` event is the last major event the browser fires during a webpage load. It signals that the browser has loaded *all* images, stylesheets, and scripts. Usually (though not necessarily) the page is stable by this point and doesn't change. We can say that when `load` has executed, the page is done loading. However, in reality, the two times above are much more important for a user's perception of page loads. [Above-the-fold render time is so Web 2.0](#).

Our optimal `head` tag will try to optimize *all* of these times. It's important to note that often you'll be presented with a tradeoff - you can decrease time to first paint by increasing time to load, and vice versa. I'm going to point out these tradeoffs, but generally I'm going to prefer to decrease time to first text paint.

Encoding

Here's an easy optimization to start us off. When a browser downloads your page off the network, it's just a stream of bits and bytes, and the browser doesn't really know what character encoding you used. Before it can read the data, it needs to decide on a

character encoding to use to read the document. 99.9% of the time on the web, we do this with UTF-8, but that isn't guaranteed.

The browser has to decide what *character encoding to use*. There's a couple of ways it can do this (fastest first):

- **The content-Type HTTP header** By putting the document's character encoding right in the response headers, you're ensuring that the browser sets the right character encoding before it even tries to parse the document. This is perfect.
- **meta tag** This is probably the most common option. For example, [Bootstrap's example page does this](#). If you do this, it's important that it's the first element in the `head`. If the browser starts reading the document with a different encoding (old IE will sometimes use some weird Windows encoding), it has to go back to the beginning and restart.
- **Guessing** If there's no `meta` tag, and no HTTP header, the browser will try to guess, using things like byte ordering characters. Of course, there are obvious compatibility issues there (and only God knows what old IE will guess), but it's also probably the slowest of all the options.

`X-UA-Compatible` is similar to character encoding - we want as high up in the document as possible because if you specify a value that's different than what the browser is already using to parse the document, you'll restart the rendering process. If you have to specify a `X-UA-Compatible` value, here's some tips:

- If you can, specify `X-UA-Compatible` in an HTTP header, not in the document itself. This is faster for the same reasons as it is for character encoding, above.
- If it has to be in the document, put `X-UA-Compatible` as high up as you can, specifically within the first 4KB of the response. IE10 and above will [speculatively prescan the first 4KB of the document](#) looking for an `X-UA-Compatible` tag. Putting it lower on the page will cause page rendering to *stop and restart*. Ouch.

Viewports

Here's another one. If you're going to specify a `viewport` size, do it at the top of the `head`.

Why?

Browsers translate this:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

...into this:

```
<style>
@viewport {
  zoom: 1.0;
  width: device-width;
}
</style>
```

While [the spec for how this works is still unfinished](#), you can bet that most browsers already implement it this way.

There's a problem with this - if you put the `viewport` meta tag *after* your stylesheets, you will cause [a layout reflow](#) for the entire document, slowing down rendering. Don't do that. Keep your viewport tags at the top, right after your character encoding. In addition, putting a viewport tag at the bottom of the head will almost certainly cause a "flash of unstyled content" as the CSS is first loaded in the default viewport, then re-rendered in your specified viewport.

Concatenation of Assets

TCP isn't really designed for short bursts. It's got a load of overhead, and needs a lot of back-and-forth just to set up a connection.

Despite this, the top 1000 websites in the world *on average* require 31-40 TCP connections. I'm sure all of them are important, and aren't [advertisements](#), [creepy 3rd-party trackers](#), or [bloatware](#)! Surely, all of those requests are for sub-resources and not a single one could be eliminated.

Alright, jokes aside, here's the scoop. Opening a new TCP connection is slow - it's especially slow if you're asking for content from a different domain (you might need to resolve DNS, negotiate TLS, and more). Minimize new connections where you can. One of the easiest places to do this is by concatenating your assets.

Although the Rails asset pipeline has been a constant source of headache for beginner Rails developers, it is absolutely one of the best performance optimizations that the framework provides.

Concatenate all of your site's stylesheets and scripts into one file each. It's 2015. There's no excuse. Yes, I know all of this will change when HTTP2 becomes widespread. But it isn't yet, and might not be for at least another year or two.

If you've got a lot of images, it may be time to start thinking about image sprites or an icon font.

All of this can be benchmarked in the wonderful Chrome Network tab - try different configurations and watch the results.

Async Defer

I'm a Ruby guy, but I hear those Javascript people talking about "async" stuff a lot. It seems like the cool thing these days - everything is "asynchronous" and "non-blocking"! But I live in Ruby land, and most things in our applications are synchronous and blocking. Gee, thanks GIL.

Ordinarily, script tags with an external `src` attribute (that is, not inlined) are synchronous and blocking too.

```
<script type="text/javascript" src="//some.shitty.thirdpartymarketingsite.com/crap
tracker.js"></script>
```

When this tag is in the head, the browser *cannot proceed with rendering the page* until it has *downloaded* and *executed* the script. This can be slow, and even if it isn't, if you do it 6-12 times on one page it will be slow anyway (thanks TCP!). [Here's an example you can test in your own browser](#). Ouch, right? While the browser cannot proceed with rendering the page (and therefore painting anything to the screen) until it's finished executing the script, it CAN download other resources further on in the document. This is accomplished with the browser preloader.

You may be thinking this is rather ridiculous - why should a browser stop completely when it sees an external script tag? Well, thanks to The Power of Javascript, that external script tag *could* potentially wreak havoc on the document if it wanted. Heck, it could completely erase the entire document and start over with `document.write()`. The browser just doesn't know. So rather than keep moving, it has to wait, download, and execute. [This is all in the spec, if you feel like a little light reading](#).

However, in the world of front-end performance, I'm not so restricted! This is not the only way! There's an `async` attribute that can be added to any `script` tag, like so:

```
<script type="text/javascript" async src="//some.shitty.thirdpartymarketingsite.com/craptracker.js"></script>
```

And *bam!* instantly that entire Javascript file is made ***magically asynchronous*** right?

Well, no.

The `async` tag just tells the browser that this particular script *isn't required to render the page*. This is perfect for most 3rd-party marketing scripts, like Google Analytics or Gaug.es. In addition, if you're really good (and you're not a Javascript single-page-app), you may be able to make every single external script on your page `async`.

`async` downloads the script file without stopping parsing of the document - the script tag is no longer *synchronous* with the

There's also this `defer` attribute, which has slightly different effects. What you need to know is that Internet Explorer 9 and below doesn't support `async`, but it does support `defer`, which provides a similar functionality. It never hurts to just add the `defer` attribute after `async`, like so:

```
<script type="text/javascript" async defer src="//some.shitty.thirdpartymarketingsite.com/craptracker.js"></script>
```

That way IE9 and below will use `defer`, and everyone who's using a browser from after the Cold War will use `async`.

[Here's a great visual explanation of the differences between `async` and `defer`.](#)

So add `async defer` to every script tag that isn't required for the page to render.

The caveat is that there's no guarantee as to the order that these scripts will be evaluated in when using `async`, or even when they'll be evaluated. Even `defer`, which is *supposed* to execute scripts in order, sometimes won't (bugs, yay). `Async` is hard.

Stylesheets first

You may have a few non-`async` script tags remaining at this point. Webfont loaders, like Typekit, are a common one - we need fonts to render the page. Some *really* intense marketing JS, like Optimizely, should probably be loaded before the page renders to avoid any flashes of unstyled content as well.

Put any CSS before these blocking script tags.

```
<head>
  <link rel="stylesheet" media="screen" href="/assets/application.css">
  <script src="//use.typekit.net/abcde.js" type="text/javascript"></script>
```

There's no `async` for stylesheets. This makes sense - we need stylesheets to render the page. But if we put CSS (external or inlined) after an external, blocking script, the browser can't use it to render the page until that external script has been downloaded and executed.

This may cause flashes of unstyled content. The most common case is the one I gave above - web fonts. A great way to manage this is with CSS classes. While loading web fonts with Javascript, TypeKit (and many other font loaders) apply a CSS class to the body called `wf-loading`. When the fonts are done loading, it changes to `wf-active`. So with CSS rules like the below, we can hide the text on the page until we've finished loading fonts:

```
.wf-loading p {
  visibility: hidden;
}
```

While text is the most important part of a webpage, it's better to show some of the page (content blocks, images, background styles) than none of it (which is what happens when your external scripts come before your CSS).

Checklist for Your App

- **Specify content encoding with HTTP headers where possible.** Otherwise, do it with meta tags at the top of the document.
- **If using `X-UA-Compatible`, put that as far up in the document as possible.**
- **`<meta name="viewport" ...>` tags should go right below any encoding tags.**

Lab: The Optimal Head Tag

Exercise 1

Let's optimize the head tag of openstreetmap.org.

The following is an abbreviated version of the head tag of OpenStreetMap.org, an open-source Rails application:

```
<html lang="en-US" dir="ltr">
  <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
  <meta name="viewport" content="width=device-width, minimum-scale=1.0, maximum-scale=1.0"/>
  <!--[if lt IE 9]><script src="/assets/html5shiv-eb6ff987ed3e1a6f2dba5ec4141d84074139c02feec781457f60a3506055b8c2.js"></script><![endif]-->
  <script src="/assets/application-4cc8181acc0e1fcd15c9aabb754b4cb2faebff5fbee02d08ac337e8c18aaa80.js"></script>
  <link rel="stylesheet" media="screen" href="/assets/screen-ltr-da6d80d62a1e392bc97642b4911053102bd3c3a5cfbc6e127b1045b8209817ac.css" />
  <link rel="stylesheet" media="print" href="/assets/print-ltr-f0e982d0ba074e914962563e26dfc48b3d2d9a05e5442932889ceec8769cd2eaf.css" />
  <link rel="stylesheet" media="screen, print" href="/assets/leaflet-all-6a764748ecb320fb225ae17114fe6a580c88369a0440e02bdac32ac203febdea.css" />
  <!--[if IE]>
    <link rel="stylesheet" media="screen" href="/assets/large-ltr-1cf527cfc2440d0b93cd4bf36e61bd90.css" />
  <![endif]-->
  <meta name="description" content="OpenStreetMap is the free wiki world map." />
  <script src="/assets/index-31bba8593ce3fd4ccd6c585180f31149973b722e3839d9a1e294fdc406c86b6e.js"></script>

  <meta name="csrf-param" content="authenticity_token" />
  <meta name="csrf-token" content="xxB3zptEZrwNUeztoP4cCeNEGd+M4krrV32BarWQ3KPARozhEdvWXlWrLuZ78nbLZZXBgqREYzoBBuFiVp1Q==" />
  <title>OpenStreetMap</title>
</head>
```

What problems do you see? How could it be improved?

Solution

- `application.js` appears before any stylesheets. This delays the evaluation of those stylesheets until after `application.js` is downloaded and executed. `application.js` should be moved to the end of the `head` tag, or marked as `async` if possible.
- No content encoding specified - add a `<meta charset="utf-8">` tag above the X-UA-Compatible tag.
- There are three CSS files here - they should all be concatenated into a single CSS file.

Resource Hints and Fighting Page Weight

There's one universal law of front-end performance - **less is more**. Simple pages are fast pages. We all know this - it isn't controversial. Complexity is the enemy.

And yet, it's trivial to find a website whose complexity seems to reach astronomical levels. Literally. The Apollo Guidance Computer had just 64 KB of ROM, but most webpages require more than 1MB of data to render. There are some webpages that are actually 100x as complex as the software that took us to the moon. It's perhaps telling that media and news sites tend to be the worst here - most media sites in 2015 take ages to load, not to mention all the time you spend clicking past their paywall popups (NYTimes) or full-page advertisements (Forbes).

Remember when [Steve Jobs said Apple's mobile products would never support Flash?](#) For a year or two there, it was a bit of a golden age in web performance - broadband was becoming widespread, 4G started to come on the scene, and, most importantly, websites started dropping Flash cruft. The "loading!" screens and unnecessarily complicated navigation schemes became something of yesteryear.

That, is, until the marketing department figured out how to use Javascript. The Guardian's homepage sets advertising tracking cookies across 4 different partner domains. Business Insider thought to one-up their neighbors across the pond and sets **cookies across 17 domains**, requires **284 requests** (to nearly 100 unique domains) and a **4.9MB download** which took a full 9 seconds to load on my cable connection, which is a fairly average broadband ~20 megabit pipe. Business Insider is, ostensibly, a news site. The purpose of the Business Insider is to deliver text content. Why does that require 5 MB of *things which are not text*?

Unfortunately, it seems, the cry of "complexity is the enemy!" is lost on the ones setting the technical agenda. While trying to load every single tracking cookie possible on your users, you've steered them away by making your site slow on *any* reasonable broadband connection, and nearly *impossible* on any mobile connection.

Usually, the boogeyman that gets pointed at is *bandwidth*: users in low-bandwidth areas (3G, developing world) are getting shafted.

But the math doesn't *quite* work out. Akamai puts the global connection speed average at **3.9 megabits per second**. So wait a second - why does Business Insider take 9 seconds to load on my 20 megabit pipe, when it's only 4.9MB? If I had an average connection, according to Akamai, shouldn't Business Insider load in 2 seconds, tops?

The secret is that "page weight", broadly defined as the simple total file size of a page and all of its sub-resources (images, CSS, JS, etc), isn't the problem. **Bandwidth is not the problem, and the performance of the web will not improve as broadband access becomes more widespread.**

The problem is latency.

Most of our networking protocols require a lot of round-trips. Each of those round trips imposes a latency penalty. Network latency is a kind of speed limit imposed on our network traffic, governed by the speed of light. Which means that latency *isn't going anywhere*.

DNS lookup is, and always will be, expensive. In 10 years, we may have invented some better protocols here. But it's fair to say we have to live with the current reality for at least a decade. Look at how long it's taking us to get on board with IPv6.

TCP connections are, and always will be, expensive.

SSL handshakes are, and always will be, expensive. We're going to be doing more of them over the next 10 years. Thanks NSA.

Each of these things requires at least one *network round-trip* - that is, a packet going from your computer, across the network, to someone else's. That will never be faster than the speed of light - and even light takes 30 milliseconds to go from New York to San Francisco and back. Thanks to the amount of hops a packet has to make across the internet backbone, usually the time is much worse - 2-4x. What's worse is that these network round-trips must happen sequentially - we have to know the IP address before we start the three-way handshake for TCP, and we have to establish a TCP connection before we can start to negotiate SSL.

Setting up a typical HTTPS connection can involve *5.5 round-trips*. That's like 165 milliseconds per connection *on a really really good day*. Usually it's better than this in the US because of CDNs. But 150ms per connection isn't a bad rule of thumb - and on mobile it's much worse, closer to 300.

The smart ones among you may already see the solution - well, Nate, 165 milliseconds per connection isn't a problem! We'll just parallelize the connections! Boom! 100 connections opened in 165 milliseconds!

The problem is that HTML *doesn't work this way by default*.

We'd like to imagine that the way a webpage loads is this:

1. Browser opens connection to `yoursite.com`, does DNS/TCP/SSL setup.
2. Browser downloads the document (HTML).
3. As soon as the browser is done downloading the document, the browser starts downloading all the document's sub resources *at the same time*.
4. Browser parses the document and fills in the necessary sub resources once they've been downloaded.

Here's what actually happens:

1. Browser opens connection to `yoursite.com`, does DNS/TCP/SSL setup.
2. Browser downloads the document (HTML).
3. Browser starts parsing the document. When the parser encounters a sub-resource, it opens a connection and downloads it. If the sub-resource is an external script tag, the parser stops, waits until it the script has downloaded, executes the entire script, and then moves on.
4. As soon as the parser stops and has to wait for an external script to download, it sends ahead something called a *preloader*. The preloader *may* notice and begin downloading resources *if* it understands how to (hint: a popular Javascript pattern prevents this).

Thanks to these little wrinkles, web page loads often have new connections opening *very* late in a page load - right before the end even! Ideally, the browser would open all of those connections like in our first scenario - immediately after the document is downloaded. We want to maximize network utilization across the life of the webpage load process.

There's four ways to accomplish this:

- **Don't stop the parser.**
- **Get out of the browser preloader's way.**
- **Use HTTP caching - but not *too* much.**
- **Use the Resource Hint API.**

Glossary

I'm going to use a couple of terms here and I want to make sure we're all on the same page.

- **Connection** - A "connection" is one TCP connection between a client (your browser) and a server. These connections can be re-used across multiple requests through things like [keep-alive](#).
- **Request** - A browser "requests" resources via HTTP. 99% of the time when we're talking about requesting resources, we're talking about an HTTP GET. Each request needs to use a TCP connection, though not necessarily a unique or new one (see [keep-alive](#)).
- **Sub-resources** - In browser parlance, a sub-resource is generally any resource required to completely load the main resource (in this case, the document). Examples of sub-resources include external Javascript (that is, `script` tags with a `src` attribute), external CSS stylesheets, images, favicons, and more.
- **Parser** - When a browser tries to load your webpage, it uses a parser to read the document and decide what sub resources need to be fetched and to construct the DOM. The parser is responsible for getting the document to one of the first important events during a page load, `DOMContentLoaded`.

Letting the Preloader do its Job

Sometimes the parser has to stop and wait for an external resource to download - 99% of the time, this is an external script. When this happens, the browser starts something called a preloader. The preloader is a bit like a "parser-lite", but rather than construct the DOM, the preloader is more like a giant regex that searches for sub resources to download. If it finds a sub-resource (say an external script at the end of the document), it will start downloading it *before* the parser gets to it.

You may be thinking this is rather ridiculous - why should a browser stop completely when it sees an external script tag? Well, thanks to The Power of Javascript, that external script tag *could* potentially wreak havoc on the document if it wanted. Heck, it could completely erase the entire document and start over with `document.write()`. The browser just doesn't know. So rather than keep moving, it has to wait, download, and execute.

Browser preloaders were a huge innovation in web performance when they arrived on the scene. Completely unoptimized webpages could speed up by 20% or more just thanks to the preloader fetching resources!

That said, there are ways to help the preloader and there are ways to hinder it. We want to help the preloader as much as possible, and sometimes we want to stay the hell out of its way.

Stop inserting scripts with "async" script-injection

The marketing department says you need to integrate your site with SomeBozoAdService. They said it's really easy - you just have to "add five lines of code!". You go to SomeBozoAdService's developer section, and find that they tell you to insert this into your document somewhere:

```
var t = document.createElement('script');
t.src = "//somebozoadservice.com/ad-tracker.js";
document.getElementsByTagName('head')[0].appendChild(script);
```

There are other problems with this pattern (it blocks page rendering until it's done, for one), but here's one really important one - browser preloaders can't work with this. Preload scanners are very simple - they're simple so that they can be fast. And when they see one of these async-injected scripts, they just give up and move on. So your browser can't download the resource until the main parser thread gets to it. Bummer! It's far better to use `async` and `defer` attributes on your script tags instead, to get this:

```
<script src="//somebozoadservice.com/ad-tracker.js" async defer></script>
```

Kaboom! There are some other advantages to `async` that I get into in [this other post here](#), but be aware that one of them is that the browser preloader can get started downloading this script before the parser even gets there.

Here's a list of other things that generally don't work with browser preloaders:

- IFrames. Sometimes there's no way around using an iframe, but if you have the option - try not to. The content of the frame can't be loaded until the parser gets there.
- @import. I'm not sure of anyone that uses @import in their production CSS, but don't. Preloaders can't start fetching @import ed stylesheets for you.
- Webfonts. Here's an interesting one. I could write a whole article on webfont speed

(I should/will!), but they usually aren't preloaded. This is fixable with resource hints (we'll get to that in a second).

- HTML5 audio/video. This is also fixable with resource hints.

I've heard that in the past, preloaders wouldn't scan the body tag when blocked in the head. If that was ever true, it is no longer true in Webkit based browsers.

In addition, modern preloaders are smart enough not to request resources that are already cached. Speaking of HTTP caching...

HTTP caching

The fastest HTTP request is the one that is never made. That's really all HTTP caching is for - preventing unnecessary requests. Cache control headers are really for telling clients "Hey - this resource, it's not going to change quickly. Don't ask me again for this resource until..." That's awesome. We should do that everywhere possible.

Yet, [the size of the resource cache is smaller than you might think](#). Here's the default disk cache size in modern browsers:

| Browser | Cache Size (default) |
|----------------------|----------------------|
| Internet Explorer 9+ | ~250MB |
| Chrome | 200MB |
| Firefox | 540MB |
| Mobile Safari | 0 |
| Android (all) | ~25-80 MB |

Not as large as you might imagine. And you read that right - Mobile Safari does not have a persistent, on-disk cache.

Most browser resource caches work on an LRU basis - last recently used. So if something doesn't get used in the cache, it's the first thing to be evicted if the cache fills up.

A pattern I've often seen is to use 3rd-party, CDN-hosted copies of popular libraries in an attempt to leverage HTTP caching. The idea is to use Google's copy of JQuery (or what have you), and a prospective user to your site will already have it downloaded before

coming to yours. The browser will notice it's already in their cache, and not make a new request. There's some other benefits, but I want to pick on this one.

This sounds good in theory, but given the tiny size of caches, I'm not sure if it really works in practice. Consider how few sites actually use Google-hosted (or Cloudflare-hosted, or whatever) JQuery. Even if they did - how often is your cached copy pushed *out* of the cache by other resources? Do you know?

Consider the alternative - bundling JQuery into your application's concatenated "application.js" file (Rails' default behavior).

In the best case, the user already has the 3rd-party CDN-hosted JQuery downloaded and cached. The request to go and get your application.js doesn't take *quite* as long because it's ~20kb smaller now that it doesn't include JQuery. But remember what we said above - bandwidth is hardly the issue for most connections (saving 20kb is really saving <100ms, even on a 2MB/s DSL connection).

But consider the worst case scenario - the user doesn't have our 3rd-party JS downloaded already. Now, compared to the "stock" application.js scenario, you have to make an additional new connection to a new domain, likely requiring SSL/TLS negotiation. Without even downloading the script, you've been hit with 1-300ms of network latency. Bummer.

Consider how much worse this gets when you're including more than 1 library from an external CDN. God forbid that the script tags aren't `async`, or your user will be sitting there for a while.

In conclusion, 3rd-party hosted Javascript, while a good idea and, strictly speaking, faster in the best-case scenario, is likely to impose a huge performance penalty to users that don't have every single one of your 3rd-party scripts cached already. Far preferable is to bundle it into a single "application.js" file, served from your own domain. That way, we can re-use the already warm connection (as long you allowed the browser to "keep-alive" the connection it used to download the document) to download all of your external Javascript in one go.

Resource hints

There's another way we can maximize network utilization - through something called *resource hints*. There are couple of different kinds of resource hints. In general, most of them are telling the browser to *prepare some connection or resource in advance* of the

parser getting to the actual point where it needs the connection. This prevents the parser from blocking on the network.

- **DNS Prefetch** - Pretty simple - tell the browser to resolve the DNS of a given hostname (`example.com`) as soon as possible.
- **Preconnect** - Tells the browser to open a connection as soon as possible to a given hostname. Not only will this resolve DNS, it will start a TCP handshake and perform TLS negotiation if the connection is SSL.
- **Prefetch** - Tells to browser to download an entire resource (or sub-resource) that may be required later on. This resource can be an entire HTML document (for example, the next page of search results), or it can be a script, stylesheet, or other sub-resource. The resource is only downloaded - it isn't parsed (if script) or rendered (if HTML).
- **Prerender** - One of these things is not like the other, and prerender is it. Marking an `<a>` tag with `prerender` will actually cause the browser to get the linked `href` page and *render it before the user even clicks the anchor!* This is the technology behind Google's Instant Pages and Facebook's Instant Articles.

It's important to note that all of these are *hints*. The browser may or may not act upon them. Most of the time, though, they will - and we can use this to our advantage.

Browser support: I've detailed which browsers support which resource hints (as of November 2015) below. However, any user agent that doesn't understand a particular hint will just skip past it, so there's no harm in including them. Most resource hints enjoy >50% worldwide support (according to caniuse.com) so I think they're definitely worth including on any page.

Let's talk about each of these items in turn, and when or why you might use each of them:

DNS Prefetch

```
<link rel="dns-prefetch" href="//example.com">
```

In case you're brand new to networking, here's a review - computers don't network in terms of domain names. Instead, they use IP addresses (like `192.168.1.1` , etc). They *resolve* a hostname, like `example.com` , into an IP address. To do this, they have to go to a DNS server (for example, Google's server at `8.8.8.8`) and ask: "Hey, what's the IP

address of `some-host.com` ?" This connection takes time - usually somewhere between 50-100ms, although it can take much longer on mobile networks or in developing countries (500-750ms).

When to Use It: But you may be asking - why would I ever want to resolve the DNS for a hostname and *not actually connect to that hostname*? Exactly. So forget about `dns-prefetch`, because its cousin, `preconnect`, does exactly that.

Browser Support: Everything except IE 9 and below.

Preconnect

```
<link rel="preconnect" href="//example.com">
```

A preconnect resource hint will hint the browser to do the following:

- Resolve the DNS, if not done already (1 round-trip)
- Open a TCP connection (1.5 round-trips)
- Complete a TLS handshake if the connection is HTTPS (2-3 round-trips)

The only thing it won't do is actually download the (sub)resource - the browser won't start loading the resource until either the parser or preloader tries to download the resource. This can eliminate up to 5 round-trips across the network! That can save us a heck of a lot of time in most environments, even fast home Wifi connections.

When to Use It: Here's an example from [Rubygems.org](https://www.rubygems.org).

Taking a look at how Rubygems.org loads in [webpagetest.org](https://www.webpagetest.org), we notice a few things. What we're looking for is network utilization after the document is downloaded - once the main "/" document loads, we should see a bunch of network requests fire at once. Ideally, they'd all fire off at this point. In a perfect world, network utilization would look like a flat line at 100%, which then stops as soon as the page loads completely. Preconnect helps us to do that by allowing us to move some network tasks earlier in the page load process.

Notice these these two resources, closer to the end of the page load:



Two are related to `gaug.es`, an analytics tracking service, and the other is a GIF from a Typekit domain. The green bar here is time-to-first-byte - time spent waiting for a server response. But note how the analytics tracking service and the Typekit GIF have teal, orange, and purple bars as well - these bars represent time spent resolving DNS, opening a connection, and negotiating SSL, respectively. By adding a `preconnect` tag to the head of the document, we can move this work to the beginning of the page load, so that when the browser needs to download these resource it has a pre-warmed connection. That loads each resource ~200ms faster in this case.

You may be wondering - why hasn't the preloader started loading these resources earlier? In the case of the `gaug.es` script, it was loaded with an `"async"` script-injection tag. This is why that method is a bit of a stinker. For more about why script-injection isn't a great idea, see Ilya Grigorik's post on the topic. So in this case, rather than adding a `preconnect` tag, I'll simply change the `gaug.es` script to a regular script tag with an `async` attribute. That way, the browser preloader will pick it up and download it as soon as possible.

In the case of that Typekit gif, it was also script-injected into the bottom of the document. A `preconnect` tag would speed up this connection. However, `p.gif` is [actually a tracking beacon for Adobe](#), so I don't think that speeding that up will provide any performance benefit to the user.

In general, `preconnect` works best with sub resources that are script-injected, because the browser preloader cannot download these resources. Use [webpagetest.org](#) to seek out sub resources that load late and trigger the DNS/TCP/TLS setup cost.

In addition, it works well for script-injected resources with dynamic URLs. You can set up a connection to the domain, and then later use that connection to download a dynamic resource (like the Typekit example above). See the W3C spec:

The full resource URL may not be known until the page is being constructed by the user agent - e.g. conditional loading logic, UA adaptation, etc. However, the origin from which one or more of these resources will be fetched is often known ahead of time by the developer or the server generating the response. In such cases, a `preconnect` hint can be used to initiate an early connection handshake such that when the resource URL is determined, the user agent can dispatch the request without first blocking on connection negotiation.

Browser Support: Unfortunately, `preconnect` is probably the least-supported resource hint. It only works in very modern Chrome and Firefox versions, and is coming to Opera soon. Safari and IE don't support it.

Prefetch

```
<link rel="prefetch" href="//example.com/some-image.gif">
```

A prefetch resource hint will hint the browser to do the following:

- Everything that we did to set up a connection in the `preconnect` hint (DNS/TCP/TLS).
- But in addition, the browser will also *actually download the resource*.
- However, `prefetch` only works for resources required by *the next navigation*, not for the *current page*.

When to Use It: Consider using `prefetch` in any case where you have a good idea what the user might do next. For example, if we were implementing an image gallery with Javascript, where each image was loaded with an AJAX request, we might insert the following prefetch tag to load the next image in the gallery:

```
<link rel="prefetch" href="//example.com/gallery-image-2.jpg">
```

You can even prefetch entire pages. Consider a paginated search result:

```
<link rel="prefetch" href="//example.com/search?q=test&page=2">
```

Browser Support: IE 11 and up, Firefox, Chrome, and Opera all support `prefetch`. Safari and iOS Safari don't.

Prerender

Prerender is prefetch on steroids - instead of just downloading the linked document, it will actually pre-render the entire page! This means that pre rendering only works for HTML documents, not scripts or other sub-resources.

This is a great way to implement something like Google's Instant Pages or Facebook's Instant Articles.

Of course, you have to be careful and considerate when using prefetch and prerender. If you're prefetching something on your own server, you're effectively adding another request to your server load for every prefetch directive. A prerender directive can be

even more load-intensive because the browser will also fetch all sub resources (CSS/JS/images, etc), which may also come from your servers. It's important to only use prerender and prefetch where you can be pretty certain a user will actually use those resources on the next navigation.

There's another caveat to prerender - like all resource hints, pretenders are given much lower priority by the browser and aren't always executed. [Here's straight from the spec.](#)

The user agent may:

- Allocate fewer CPU, GPU, or memory resources to pre rendered content.
- Delay some requests until the requested HTML resource is made visible - e.g. media downloads, plugin content, and so on.
- Prevent pre rendering from being initiated when there are limited resources available.

Browser Support: IE 11 and up, Chrome, and Opera. Firefox, Safari and iOS Safari don't get this one.

Conclusion

We have a long way to go with performance on the web. I scraped together a little script to check the Alexa Top 10000 sites and look for resource hints - here's a quick table of what I found.

| Resource Hint | Prevalence |
|---------------|------------|
| dns-prefetch | 5.0% |
| preconnect | 0.4% |
| prefetch | 0.4% |
| prerender | 0.1% |

So many sites could benefit from liberal use of some or all of these resource hints, but so few do. Most sites that do use them are just using `dns-prefetch`, which is practically useless when compared to the superior `preconnect` (how often do you really want to know the DNS resolution of a host and then *not* connect to it?).

I'd like to back off from the flamebait-y title off this article *just* slightly. Now that I've explained all of the different things you can do to increase network utilization during a webpage load, know that 100% utilization isn't always possible. Resource hints and the

other techniques in this article *help* complex pages load faster, but thanks to many different constraints you may not be able to apply them in all situations. Page weight *does* matter - a 5MB page will be more difficult to optimize than a 500 KB one. What I'm really trying to say is that page weight *only sorta* matters.

I hope I've demonstrated to you that page weight - while certainly *correlated* with webpage load speed, is not the final answer. You shouldn't feel like your page is doomed to slowness because The Marketing People need you to include 8 different external ad tracking services (although you should consider quitting your job if that's the case).

Checklist for Your App

- **Reduce the number of connections required *before* reducing page size.**
Connections can be incurred by requesting resources from a new unique domain, or by requesting more than one resource at a time from a single domain on an HTTP/1.x protocol.
- **HTTP caching is great, but don't *rely* on any particular resource being cached.**
3rd-party CDNs for resources like JQuery, etc are probably not reliable enough to provide any real performance benefit.
- **Use resource hints - especially `preconnect` and `prefetch` .**

Turbolinks - The Performance Everyone Forgot

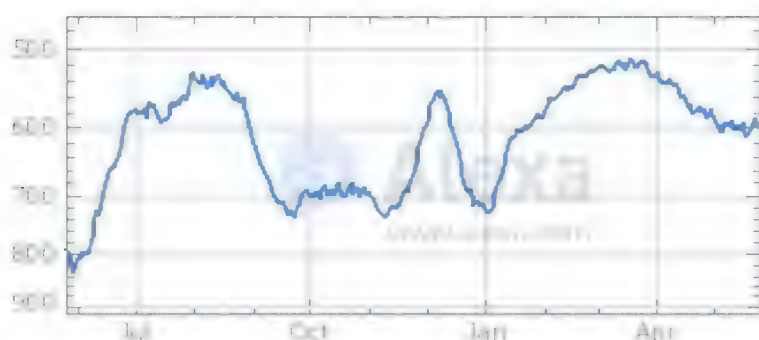
A perceived benefit of a client-side JS framework is the responsiveness of its interface - updates to the UI are instantaneous. A large amount of application logic (and, usually, state) lives on the client, instead of on the server. The client-side application can perform most tasks without running back to the server for a round-trip. As a result, in the post-V8 era, many developers think traditional server-side languages and frameworks (Ruby, Python, even Java) are simply too slow for modern web applications, which are now supposed to behave like native applications, with instantaneous responses.

Is Rails dead? Can the old Ruby web framework no longer keep up in this age of "native-like" performance?


A few days ago (May 21, 2015), Shopify IPO'd. Shopify (an e-commerce provider that lets you set up your own online shop) had [over 150,000 customers](#) and is a [Top 1000](#) site on Alexa. In addition, Shopify hosts their customers' sites, with an average of 100ms response times for over 300 million monthly page views. Now that's Web Scale. And they did it all on Rails.

Alexa Traffic Ranks

How is this site ranked relative to other sites?



Global Rank ?

 **592** 

Rank in [United States](#) ?

 **427**

They're not the only ones doing huge deployments with blazing fast response times on Rails. [DHH claims Basecamp's average server response time is 27ms](#). [Github averages about 60ms](#).

But fast response times are only half of the equation. If your server is blazing fast, but you're spending 500-1000ms on each new page load rendering the page, setting up a new Javascript VM, and re-constructing the entire render tree, your application will be fast, but it won't be *instantaneous*.

Enter [Turbolinks](#).

Turbolinks and other "view-over-the-wire" technologies

Turbolinks received (and still receives) a huge amount of flak from Rails developers upon its release. Along with [pjax](#), from which it evolved, Turbolinks represented a radical shift in the way Rails apps were intended to be architected. With hardly any effort at all, Rails apps had similar characteristics to the "Javascript single-page app" paradigm: no full page loads, `pushState` usage, and AJAX.

But there was a critical difference between Turbolinks and their SPA brethren: instead of sending *data* over the wire, Turbolinks sent *fully rendered views*. Application logic was reclaimed from the client and kept on the server again. Which meant we got to write more Ruby! I'll call this approach "views-over-the-wire", because we're sending HTML, not data.

"View-over-the-wire" technologies like turbolinks and pjax have laid mostly out of the limelight since their release in ~2012, despite their usage by such high-profile sites as [Shopify](#) and Github. But with Rails 5, Turbolinks is getting a nice upgrade, with new features like partial replacement and a progress bar with a public API. So I wanted to answer for myself the question: how does building an application with Turbolinks feel? Can it be not just fast, but *instantaneous*?

And just what is an instantaneous response? The guidelines for human-computer interaction speeds have remained constant since [they were first discovered in the late 60's](#):

- **0.1 second** is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result.
- **1.0 second** is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data.
- **10 seconds** is about the limit for keeping the user's attention focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, since users will then not know what to expect.

See the rest of their take on response times [here](#)." -->

Can Turbolinks help us achieve sub-0.1 second interaction?

In the non-Turbolinks world, Rails apps usually live in the 1.0 second realm. They return a response in 100-300ms, spend about 200ms loading the HTML and CSSOM, [a few hundred more ms rendering and painting](#), and then likely loads of JS scripting tied to the onload event.

But in the Turbolinks/pjax world, we get to cut out a lot of the work that usually happens when accessing a new page. Consider:

1. When using Turbolinks, you don't throw away your entire Javascript runtime on every page. We don't have to attach a thousand event listeners to the DOM, nor throw out any JS variables between page loads. This requires you to rethink the way you write your Javascript, but the speed benefits are big.
2. When using Turbolinks partial replacement, we don't even throw away the entire DOM, instead changing only the parts we need to change.
3. We don't have to parse and tokenize the CSS and JS ever again - the CSS Object Model is maintained.

All of this translates into eliminating 200-700ms on each new page. This lets us move out of the 1 second human-computer interaction realm, and start to flirt with the 100 ms realm of "instantaneous" interaction.

As an experiment, I've constructed a TodoMVC app using Rails 5 (still under active development) and Turbolinks 3. You can find [the application here](#) and [the code here](#). It also uses partial replacement, a new feature in Turbolinks 3. Using your browsers favorite development tools, you can confirm that most interactions in the app take about 100-250ms, from the time the click event is registered until the response is painted to the screen.

By comparison, the reference Backbone implementation for TodoMVC takes about 25-40ms. Consider also that our Backbone implementation isn't making any roundtrips to a server to update data - most TodoMVC implementations use LocalStorage. I can't find a live TodoMVC implementation that uses a javascript framework *and* a server backend, so the comparison will have to suffice. In any case, after removing network timing, Turbolinks takes about the same amount of time to update the page state and paint the new elements about as quickly as Backbone. And we didn't even have to write any new Javascript!

Turbolinks also forces you to do a lot of things you should be doing already with your front-end Javascript - idempotent functions, and not treating your DOM ready hooks like a junk drawer. A lot of people griped about this when Turbolinks came out - but you shouldn't have been doing it anyway!

Other than asking to re-evaluate the way you write your front-end JS, Turbolinks doesn't ask you to change a whole lot about the way you write Rails apps. You still get to use all the tools you're used to on the backend, because what you're doing is still The Web with a little spice thrown in, not [trying to build native applications in Javascript](#).

load is dead, all hail load!

Look in any Rails project, and for better or for worse, you're going to see a lot of this:

```
$(document).ready(function () { ... } );
```

Rails developers are usually pretty lazy when it comes to Javascript (although, most *developers* are pretty lazy). [jQuery waits for DOMContentLoaded to fire](#) before handing off execution to the function in `ready`. But Turbolinks takes `DOMContentLoaded` away from us, and [gives us a couple other events instead](#). Try attaching events to these instead, or using JQuery's `.on` to attach event handlers to the document (as opposed to individual nodes). This removal of the `load` and `DOMContentLoaded` events can wreak

havoc on existing Javascript that uses page ready listeners everywhere, and why I wouldn't recommend using Turbolinks on existing projects, and using it for greenfield only.

Caching - still a Rails dev's best friend

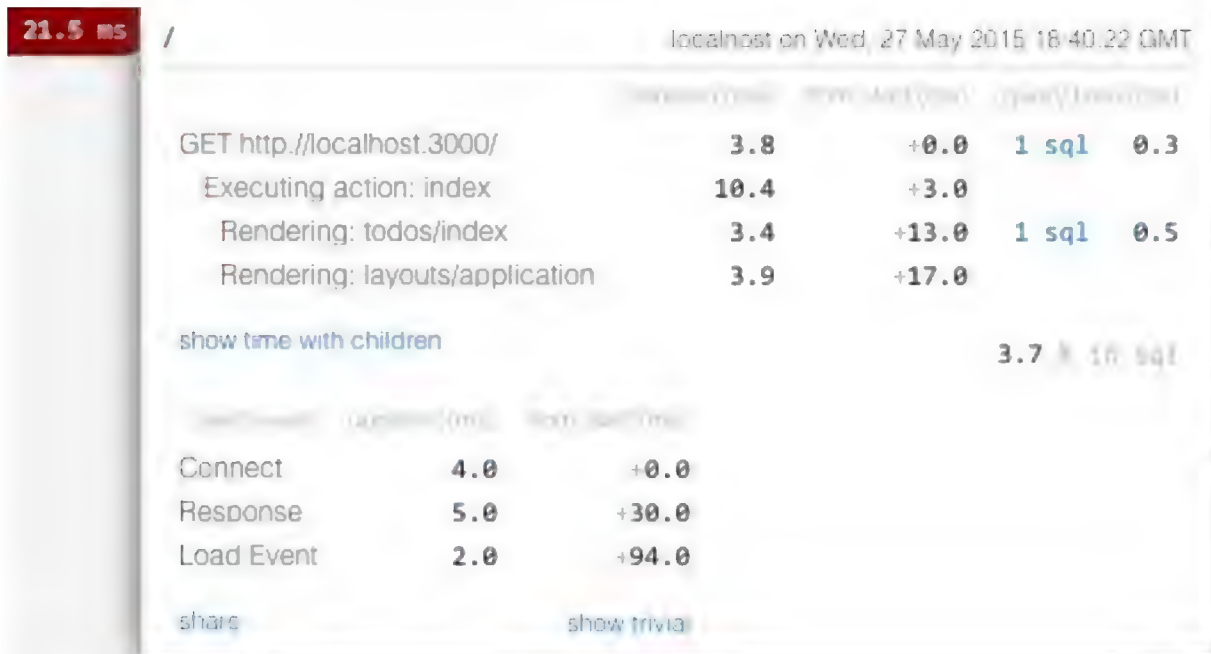
DHH has said it a hundred times: Rails is an extraction from Basecamp, and is best used when building Basecamp-like applications. Thus, DHH's [2013 talk on Basecamp's architecture](#) is valuable - most Rails apps should be architected this way, otherwise you're going to be spending most of your time fighting the framework rather than getting things done.

Most successful large-scale Rails deployments make extensive use of caching. Ruby is a (comparatively) slow language - if you want to keep server response times below 300ms, you simply have to minimize the amount of Ruby you're running on every request and never calculate the same thing twice.

Caching can be a double-edged sword in small apps, though. Sometimes, the amount of time it takes to read from the cache is more than it takes to just render something. When evaluating whether or not to cache something, *always* test your apps locally *in production mode*, with production-size datasets (just a copy of the production DB, if your company allows it). The only way to know for sure if caching is the right solution for a block of code is to measure, measure, measure. And how do we do that?

rack-mini-profiler and the flamegraph

[rack-mini-profiler](#) has become an indispensable part of my Ruby workflow. It's written by the incredible Sam Saffron, who's doing absolutely vital work (along with others) on Ruby speed over at [RubyBench.org](#).

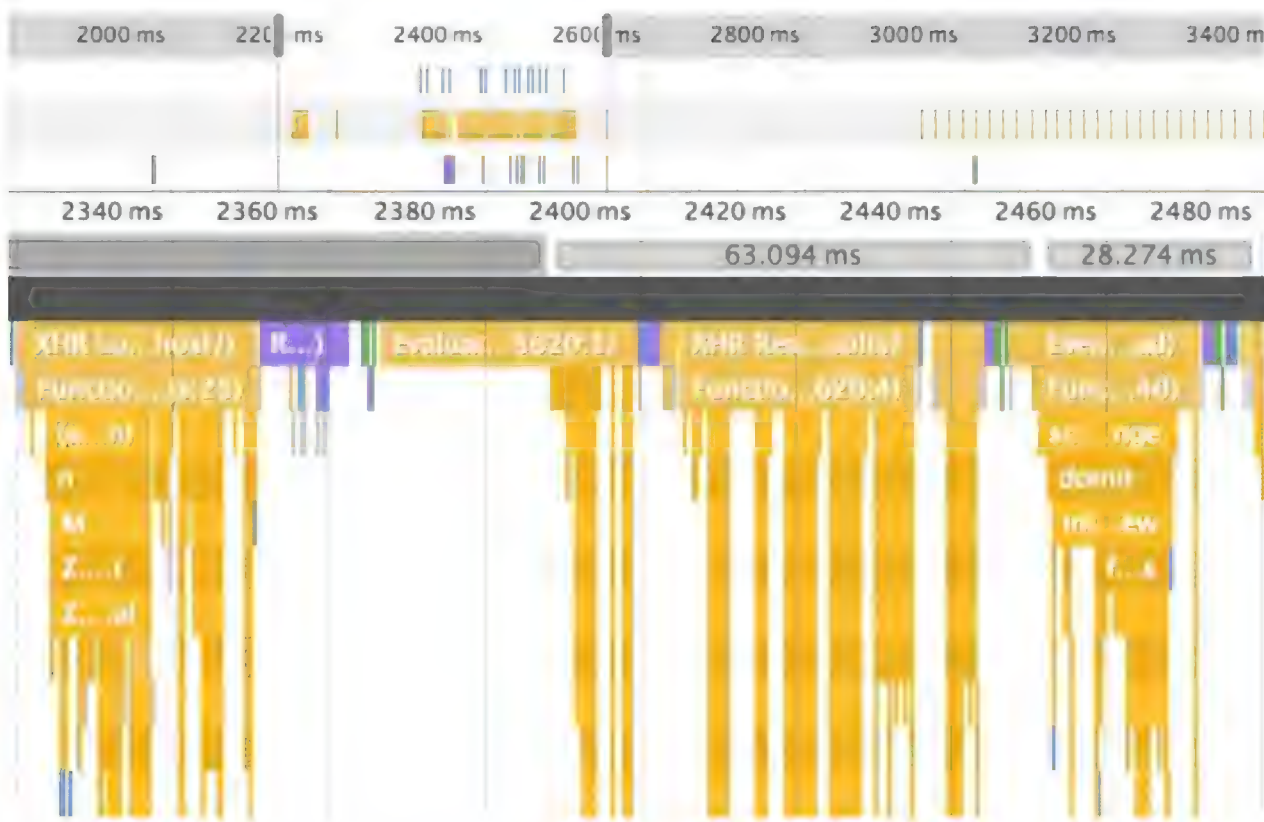


rack-mini-profiler puts a little white box at the upper left of a page, showing you exactly how long the last request took to process, along with a breakdown of how many SQL queries were executed. The amount of unnecessary SQL queries I've eliminated with this tool must number in the thousands.

But that's not even rack-mini-profiler's killer feature. If you add in the `flamegraph` gem to your Gemfile, you get a killer flame graph showing exactly how long rendering each part of your page took. This is invaluable when tracking down exactly what parts of the page took the most time to render.

Chrome Timeline - the sub-100ms developer's best friend

When you're aiming for a sub-100ms-to-glass Turbolinks app, every ms counts. So allow me introduce you to my little friend: Chrome Timeline.



This bad boy shows you, in flamegraph format, exactly where each of your 100ms goes. Read up on Google's documentation on exactly how to use this tool, and exactly what means what, but it'll give you a great idea of which parts of your Javascript are slowing down your page.

Rethinking Turbolinks 3's progress bar

Turbolinks 3 introduces a progress bar to the top of the page (you can see it in action at this demo here). When building my TodoMVC implementation, I noticed that this progress bar actually *increased* perceived load times in sub-200 ms interactions. I'm not sure what it is about it - perhaps my brain is just wired to think that an interaction is slow whenever it sees a progress bar or a spinner.

[Turbolinks exposes a public API for the progress bar](#) - eventually, someone will hack together a way to hide the progress bar until a preset amount of time has passed - say, 250ms.

Non-RESTful redirects

100ms-to-glass is *not* a lot of time. In most cases, you may not even have time to redirect. Consider this typical bit of Rails controller logic:

```
def create
  thing = Thing.new(params[:thing])
  if thing.save
    redirect_to ...
```

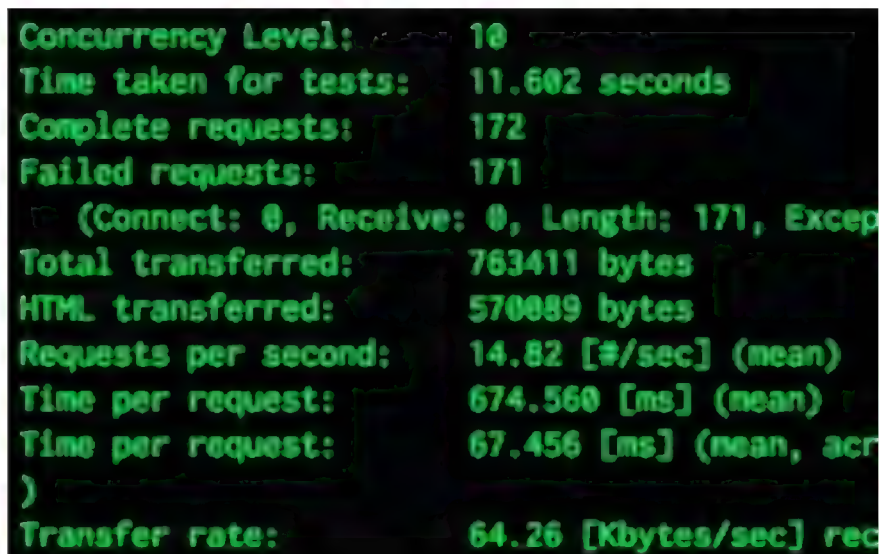
Unfortunately, you've just doubled the number of round-trips to the server - one for the POST, and one for the GET when you get your response back from the redirect. I've found that this alone puts you beyond 100ms. With remote forms and Turbolinks, it seems to be far better to do non-RESTFUL responses here and [just re-render the \(updated\) index view](#).

Be wary of partials

Partials in Rails have always been slow-ish. They're fast enough if you're aiming for 300ms responses, but in the 100ms-to-glass world, we can't really afford any less than a 50ms server response time. Be wary of using partials, cache them if you can, and always benchmark when adding a new partial.

Response time goals and Apache bench

Another key tool for keeping your Turbolinks-enabled Rails app below 100ms-to-glass is to keep your server response times ridiculously fast - 50ms should be your goal. Apache Bench is a great tool for doing this, but siege is another popular tool that does the same thing - slams your web server as fast as it can to get an idea of your max requests/second.



```
Concurrency Level:      10
Time taken for tests:    11.602 seconds
Complete requests:      172
Failed requests:         171
  (Connect: 0, Receive: 0, Length: 171, Exceptions: 0)
Total transferred:      763411 bytes
HTML transferred:       570089 bytes
Requests per second:    14.82 [#/sec] (mean)
Time per request:       674.560 [ms] (mean)
Time per request:       67.456 [ms] (mean, across all concurrent requests)
Transfer rate:          64.26 [Kbytes/sec] received
```

Be sure to load up your rails server in production mode when benchmarking with these tools so that you don't have code reloading slowing down each request!

In addition, be sure to test with production (or extremely production-like) data. If queries return 100 rows in development but return 1000 rows in production, you're going to see different performance. We want our development environment to be as similar to production as possible.

Gzip all the things!

Slap Rack::Deflater [on the tippy-top of your middleware stack](#) to gzip any asset responses, along with any HTML.

Things that don't work

I tried using JRuby, and the speed impact was negligible. Even when testing with a big concurrent load, multi-process MRI servers like Unicorn or Puma performed better. Your app may be different, but don't expect a huge speed boost here.

Turbo-React is an interesting project. It combines Turbolinks with React's virtual DOM and dom-diffing tools, meaning that Turbolinks does the least amount of work possible when changing the current document to the new one that came down the wire (by default, Turbolinks just swaps out the entire document). I gave it a shot, though, and didn't notice any meaningful speed differences. It does, however, let you do cool things like CSS transitions.

I noticed I was spending 30-40ms in scripting on each Turbolinks request. As a hunch, I tried swapping JQuery out with Zepto, to see if my choice of framework was making a difference. No luck. Zepto's just as slow as JQuery when working with Turbolinks. Also unfortunately, I can't forgo JQuery either, since to stay under 100ms I'm forced to use remote forms, which requires a javascript framework of some kind (either JQuery to use jquery-ujs, or zepto to use rails-behaviors).

Common mistakes

- Be absolutely certain that a page load that you *think* is Turbolinks enabled, is actually Turbolinks enabled. Click a link with the Developer console open - if the console says something like "Navigated to <http://www.whatever.com/foo>", that link wasn't Turbolinks-enabled.
- Don't render erb responses that do things like append items to the current page. Instead, a Turbolinks-enabled action should return a full HTML page. Let Turbolinks do the work of swapping out the document, instead of writing your own, manual

"\$("#todo-list").append("<%= j(render(@todo)) %>");" calls. For an example, [check out my TodoMVC implementation](#), which only uses an index template. Keep state (elements having certain classes, for example) in the template, rather than allowing too much DOM state to leak into your Javascript. It's just unnecessary work that Turbolinks frees us from doing.

Limitations and caveats

I'm not sure how Turbolinks will fare in more complex UI interactions - the TodoMVC example is simple. Caching *will* be required when scaling, which some people think is too complex. I think that with smart key-based expiration, and completely avoiding manual cache expiration or "sweepers", it isn't too bad.

Turbolinks doesn't play great with client side JS frameworks, due to the transition cache and the lack of the `load` event. Be wary of multiple instances of your app being generated, and be careful of Turbolinks' transition cache.

Integration testing is still a pain. Capybara and selenium-webdriver, though widely used, remain difficult to configure properly and, seemingly no matter what, are not deterministic and occasionally experience random failures.

On mobile

First, if you're interested in fast mobile sites, Ilya Grigorik's 1000ms to glass talk is required viewing. Ilya reveals that for mobile, you're going to be spending at least 300-700ms just waiting for the network. That means that, unfortunately, a Turbolinks enabled approach cannot get you to a 0.1ms instantaneous UI on mobile.

"View-over-the-wire" applications don't work offline, while client side JS applications (in theory) can.

Conclusion: "View-over-the-wire" is better than it got credit for

Overall, I quite enjoyed the Turbolinks development experience, and mostly, as a user, I'm extremely impressed with the user experience it produces. Getting serious about Rails performance and using a "view-over-the-wire" technology means that Rails apps will deliver top-shelf experiences on par with any clientside framework.

Checklist for Your App

- **Be aware of the speed impact of partials.** Use profilers like `rack-mini-profiler` to determine their real impact, but partials are slow. Iterating over hundreds of them (for example, items in a collection) may be a source of slowdown. Cache aggressively.
- **Static assets should always be gzipped.** As for HTML documents, the benefit is less clear - if you're using a reverse proxy like NGINX that can do it for you quickly, go ahead and turn that on.
- **Eliminate redirects in performance-sensitive areas.** 301 redirects incur a full network round-trip - in performance sensitive code, such as simple Turbolinks responses, it may be worth it to render straight away rather than redirect to a different controller action. This does cause some code duplication.

WebFonts, Design's Double-Edged Sword

I'm passionate about fast websites. That's a corny thing to say, I realize - it's something you'd probably read on a resume, next to a description of how "detail-oriented" and "dedicated" I am. But really, I love the web. The openness of the Web has contributed to a global coming-together that's created beautiful things like Wikipedia or the FOSS movement.

As Jeff Bezos has said to [Basecamp](#), nobody is going to wake up 10 years from now and wish their website was slower. By making the web faster, we can make bring the Web's amazing possibilities for collaboration to an even wider global audience.

Internet access is not great everywhere - the [Akamai State of the Internet, 2015](#) puts the global average connection bandwidth at 5.1 Mbps. For those of you doing the math at home, that's a measly 625 kilobytes per second. The US average isn't much better - 12.0 Mbps, or just 1.464 megabytes per second.

When designing the website for a project that wants to encourage global collaboration, as most FOSS sites do, we need to be thinking about our users in low-bandwidth areas (which is to say, the majority of global internet users). We don't want to make a high-bandwidth connection a barrier to learning a programming language or contributing to open-source.

It's with this mindset that I've been looking at the performance of [Rubygems.org](#) for the last few weeks. As a Rubyist, I want people all over the world to be able to use Ruby - fast connection or no.

Rubygems.org is one of the most critical infrastructure pieces in the Ruby ecosystem - you use it every time you `gem install` (or `bundle install`, for that matter).

Rubygems.org also has a web application, which hosts a gem index and search function. It also has some backend tools for gem maintainers.

I decided to dig in to the front-end performance of Rubygems.org for these reasons.

Diagnosing with Chrome Timeline

When diagnosing a website's performance, I do two things straight off the bat:

- Open the site in Chrome. Open DevTools, and do a hard refresh while the Network tab is open.
- Run a test on webpagetest.org.

Both webpagetest.org and Google Chrome's Network tools pointed out an interesting fact - while total page weight was reasonable (about 600 KB), over 72% of the total page size was WebFonts (434 KB!). Both of these tools were showing that page loads were being heavily delayed by waiting for these fonts to download.

I plugged Akamai's bandwidth statistics into DevTool's network throttling function. Using DevTool's throttler is a bit like running your own local HTTP proxy that will artificially throttle down network bandwidth to whatever values you desire. The results were pretty dismal. Lest you try this on your own site, don't immediately discard the results if you think they're "way too slow, our site never loads like that!" At 625 KB/s, Twitter still manages to paint within 2 seconds. Google's homepage does it half a second."

| | Time to First Paint | Time to Paint Text (fonts loaded) | Time to load Event |
|----------------------|---------------------|-----------------------------------|--------------------|
| US (1.4 MB/s) | 3.56s | 3.83s | 3.96s |
| Worldwide (625 KB/s) | 7.41s | 7.59s | 8.20s |

Ouch! I used DevTool's Filmstrip view to get a rough idea of when fonts were loaded in as well. You can use the fancy new [Resource Timing API](#) to get this value precisely (and on client browsers!) but I was being lazy.

When evaluating the results of any performance test, I use the following rules-of-thumb. These guidelines for human-computer interaction speeds have remained constant since [they were first discovered in the late 60's](#) :

- **0.1 seconds** is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result.
- **1.0 second** is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data.
- **10 seconds** is about the limit for keeping the user's attention focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the

computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, since users will then not know what to expect.

This is the Nielsen Norman group's interpretation of the linked paper. [See the rest of their take on response times.](#)

Most webpages become *usable* (that is, the user can read and begin to interact with them) in the range of 1 to 10 seconds. This is *good*, but it's possible that for many connections we can achieve websites that, on first/uncached/cold loading, can be usable in less than 1 second.

Using these rules-of-thumb, I decided we had some work to do to improve Rubygems.org's paint and loading times on poor connections. As fonts comprised a majority of the site's page weight, I decided to start there.

Auditing font usage

WebFonts are awesome - they really make the web beautiful. [Web design is 95% typography](#)", so changing fonts can have a huge effect on the character and feel of a website. For these reasons, WebFonts have become extremely popular - [HTTP Archive](#) estimates about 51% of sites currently use WebFonts and that number is still growing.

WebFonts are here to stay, but that doesn't mean it's impossible to use them poorly.

Rubygems.org was using Adobe Typekit - a common setup - and using a single WebFont, Aktiv Grotesk, for all of the site's text.

By using Chrome's Network tab, I realized that Rubygems.org was loading more than a dozen individual weights and styles of the site font, Aktiv Grotesk. Immediately some red flags started to go up - how could I possibly audit all of the site's CSS and determine if each of these weights and styles was actually being used?

Instead of taking a line-by-line approach of combing through the CSS, I decided to approach the problem from first principles - what was the intent of the design? *Why* was Rubygems.org using WebFonts?

Deciding on Design Intent

Now, I am not a designer, and I don't pretend to be one on the internet. As developers, our job isn't to tell the designers "Hey, you're dumb for including over 500KB of WebFonts in your design!". That's not their job. As performance-minded web developers, our job is to **deliver the designer's vision in the most performant way possible**.



This is a screenshot of Rubygems.org's homepage. Most of the text is set at around a ~14px size, with the notable exception of the main heading, which is set in large type in a light weight. All text is set in the same font, Aktiv Grotesk, which could be described as a grotesque or neo-grotesque sans-serif. What's a grotesque? [Wikipedia has a good description](#).

Based on my interpretation of the design, I decided the design's intent was:

- For h1 tags, use a light weight grotesque type.
- For all other text, use a grotesque type set at it's usual, context-appropriate weight.
- The design should be consistent across platforms.
- The design should be consistent across most locales/languages.

The site's font, Aktiv Grotesk, bears more than a passing resemblance to Helvetica or Arial - they're both grotesque sans-serifs. At small (~14px) sizes, the difference is mostly indistinguishable to non-designers.

I already had found a way to eliminate the majority of the site's WebFont usage - use WebFonts only for the h1 header tags. The rest of the site could use a Helvetica/Arial font stack with little visual difference. **This one decision eliminated *all but one of the weights and styles* required for Rubygems.org!**

Using WebFonts for "body" text - paragraphs, h3 and lower - seems like a loser's game to me. The visual differences to system fonts are usually not detectable at these small sizes, at least to layman eyes, and the page weight implications can be immense. Body text usually requires several styles - bold, italic, bold italic at least - whereas headers usually appear only in a single weight and style. **Using WebFonts only in a site's headers is an easy way to set the site apart visually without requiring a lot of WebFont downloads.**

I briefly considered not using WebFonts at all - most systems come with a variety of grotesque sans-serifs, so why not just use those on our headers too? Well, this would work great for our Mac users. Helvetica looks stunning in a light, 100 weight. But Windows is tougher. Arial isn't included in Windows in anything less than 400 (normal) weight, so it wouldn't work for Rubygems.org's thin-weight headers. And Linux - well, who knows what fonts they have installed? It felt more appropriate to *guarantee* that this "lightweight" header style, so important to the character of the Rubygems.org design, would be visually consistent across platforms.

So I had my plan:

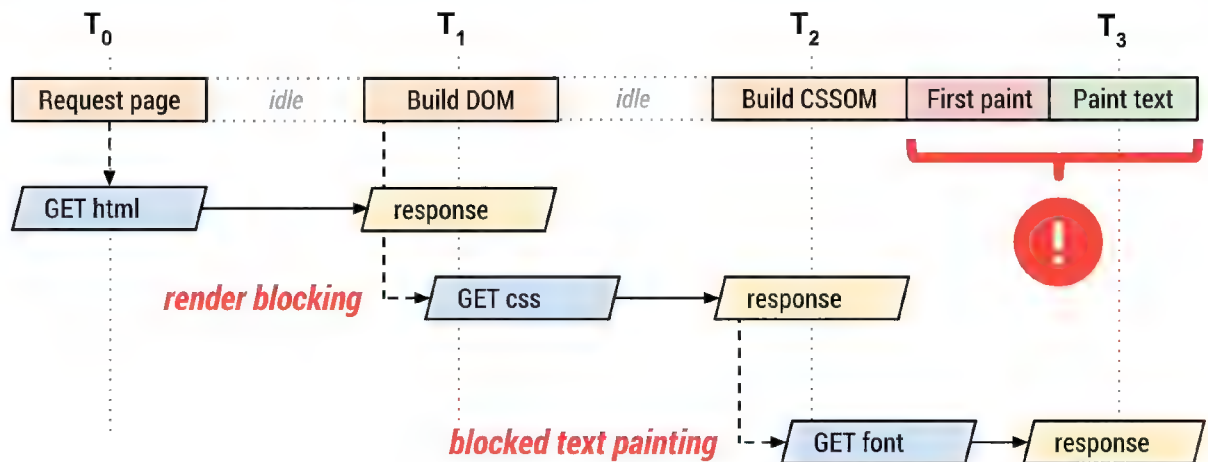
- Use a WebFont, in a grotesque sans-serif style, to display all the site's h1 tags in a light weight.
- Use the common Helvetica/Arial stack for all other text.

Changing to Google Fonts

Immediately, I knew Typekit wasn't going to cut it for Rubygems.org. Rubygems.org is an open-source project with many collaborators, but issues with fonts had to go through one person (or a cabal of a few people), the person that had access to the Typekit account. With an OSS font, or a solution like Google Fonts (where anyone can create a new font bundle/there is no 'account'), we could all debug and work on the site's fonts.

That reason - the "accountless" and FOSS nature of the fonts served by Google Fonts - initially lead me to use Google Fonts for Rubygems.org. Little did I realize, though, that Google Fonts offers a number of performance optimizations over Typekit that would end up making a huge difference for us.

Serve the best possible format for a user-agent



In contrast to Typekit, Google Fonts works with a two-step process:

- You include an external stylesheet, hosted by Google, in the head tag. This stylesheet includes all the `@font-face` declarations you'll need. The actual font files themselves are linked in this stylesheet.
- Using the URLs found in the stylesheet, the fonts are downloaded from Google's servers. Once they're downloaded, the browser renders them in the document.

Typekit uses [WebFontLoader](#) to load your fonts through an AJAX request.

When the browser sends the request for the external stylesheet, Google takes note of what user agent made the request.

But why would different browsers need different fonts served?

- **Not all font formats are created equal, and browsers require different formats.** Ideally, everyone would support and use WOFF2, the latest open standard. WOFF2 uses some awesome compression that can reduce font sizes by up to 30% over the more widely supported WOFF1. Some browsers (mostly old IE and Safari) require EOT, TTF, even SVG. Google Fonts takes care of all of this *for you*, rather than you having to host and serve each of these formats yourself.
- **Google strips out font-hinting information for non-Windows users** What's font hinting? [Via Wikipedia](#): "Font hinting (also known as instructing) is the use of mathematical instructions to adjust the display of an outline font so that it lines up with a rasterized grid. At low screen resolutions, hinting is critical for producing clear, legible text." This is pretty cool. Only Windows usually actually uses this information in a font file - Mac and other operating systems have their own "auto-hinting" that ignores most of this information. If there is any hinting information in a font file, Google will strip it out for non-Windows users, eliminating a few extra bytes of data.

Leveraging the power of HTTP caching

As I mentioned, Google Fonts are a two-step process: download the (very short) stylesheet from Google, then download the font files from wherever Google tells you.

The neat thing is that *these font files are always the same for each user agent*.

So if you go to Rubygems.org on a Mac with Chrome, and then navigate to a *different site* that uses the same Google Fonts served Roboto font and weight as we do, you *won't redownload it!* Awesome! And since Roboto is one of the most widely used WebFonts, we can be reasonably expect that at least a minority of visitors to our site *won't have to download anything at all!*

Even better, since Roboto is the default system font on Android and ChromeOS, those users won't download anything at all either! Google's CSS puts the *local* version of the font higher up in the font stack:

```
@font-face {  
  font-family: 'Roboto';  
  font-style: normal;  
  font-weight: 100;  
  src: local('Roboto Thin'), local('Roboto-Thin'), url(https://fonts.gstatic.com/s/roboto/v15/2tsd397wLxj96qwHyNIkxHYhjbSpvc47ee6xR_80Hnw.woff2) format('woff2');  
}
```

Google Font's stylesheet has a cache lifetime of 1 day - but the font files themselves have a cache lifetime of 1 year. All in all, this adds up - many visitors to Rubygems.org won't have to download any font data at all!

Removing render-blocking Javascript

One of my main beefs with Typekit (and [webfont.js](#)) is that it introduces Javascript into the critical rendering path. Remember - any time the browser's parser encounters a script tag, it must:

- Download the script, if it is external (has a "src" attribute) and isn't marked `async` or `defer`.
- Evaluate the script.

Until it finishes these two things, the browser's parser is *stuck*. It can't move on constructing the page. Rubygems.org's Typekit implementation looked like this:

```
<html lang="en-us">
  <head>
    <script src="//use.typekit.net/omu5dik.js" type="text/javascript"></script>
    <script>
      try{Typekit.load();}catch(e){}
    </script>
    <%= stylesheet_link_tag("application") %>
  </head>
```

Arrgh! We can't start evaluating this page's CSS until Typekit has downloaded itself and `Typekit.load()` has finished. Unfortunately, if, say, Typekit's servers are slow or are down, `Typekit.load()` will simply block the browser parser until it times out. Ouuccch! This could take your entire site down, in effect, if Typekit ever went down (this has happened to me before - don't be as ignorant as I!).

Far better would have been this:

```
<html lang="en-us">
  <head>
    <%= stylesheet_link_tag("application") %>
    <script src="//use.typekit.net/omu5dik.js" type="text/javascript"></script>
    <script>
      try{Typekit.load();}catch(e){}
    </script>
  </head>
```

At least in this case we can render everything *except* the WebFonts from Typekit. We'll still have to wait around for any of the text to show up until after Typekit finishes, but at least the user will see *some* signs of life from the browser rather than staring at a blank white screen.

Google Fonts doesn't use any JavaScript (by default, anyway), which makes it faster than almost any JavaScript-enabled approach.

There's really only one case where using Javascript to load WebFonts makes sense - preventing flashes of unstyled text. Certain browsers will immediately render the fallback font (the next font in the font stack) without waiting for the font to download. Most modern browser will instead wait, sensibly, for up to 3 seconds while the font downloads.

What this means is that using Javascript (really I mean `webfont.js`) to load WebFonts makes sense when:

- Your WebFonts may reasonably be expected to take more than 3 seconds to

download. This is probably true if you're loading 500KB or more of WebFonts. In that case, `webfont.js` (or similar) will help you keep text hidden for longer while the WebFont downloads.

- You're worried about FOUC in old IE or *really* old Firefox/Chrome versions. Simply keeping WebFont downloads fast will minimize this too.

unicode-range

If you look at Rubygems.org in Chrome, Safari, Firefox, and IE, you'll notice something different in the size of the font download:

| Browser | Font Format | Download Size | Difference |
|---------------|-------------|---------------|------------|
| Chrome (Mac) | WOFF2 | 10.0 KB | 1x |
| Opera | WOFF2 | 10.0 KB | 1x |
| Safari | TrueType | 62.27 KB | 6.27x |
| Firefox (Mac) | WOFF | 58.9 KB | 5.89x |
| Chrome (Win) | WOFF2 | 14.4 KB | 1.44x |
| IE Edge | WOFF | 78.88 KB | 7.88x |

What the hell? How is Chrome only downloading 10KB to display our WebFont when Safari and Firefox take almost 6x as much data? Is this some secret backdoor optimization Google is doing in Chrome to make other browsers look bad?! Well, Opera looks pretty good too, so that can't be it (this makes sense - they both use the Blink engine). Is WOFF2 just *that good*?

If you take a look at the CSS Google serves to Chrome versus the CSS served to other browsers, you'll notice a crucial difference in the `@font-face` declaration:

```
@font-face {
  font-family: 'Roboto';
  unicode-range: U+0000-00FF, U+0131, U+0152-0153, U+02C6, U+02DA, U+02DC, U+2000-
206F, U+2074, U+20AC, U+2212, U+2215, U+E0FF, U+EFFD, U+F000;
}
```

What's all this gibbledy-gook?

The `unicode-range` property *describes what characters the font supports*. Interesting, right? Rubygems.org, in particular, has to support Cyrillic, Greek and Latin Extended characters. Normally, we'd have to download extra characters to do that.

By telling the browser what characters the font supports, the browser can look at the page, note what characters the page uses, and then *only download the fonts it needs to display the characters actually on the page*. Isn't that awesome? Chrome (and Opera) isn't downloading the Cyrillic, Latin-Extended or Greek versions of this font because it knows it doesn't need to! [Here's the CSS3 spec on unicode-range for more info](#)

This particular optimization only really matters if you need to support different character sets. If you're just serving the usual Latin set, `unicode-range` can't do anything for you.

There are other ways to slim your font downloads on Google Fonts, though - there's a semi-secret `text` parameter that can be given to Google Fonts to generate a font file that only includes *the exact characters you need*. This is useful when using WebFonts in a limited fashion. This is exactly what I do on this site:

```
<link href="http://fonts.googleapis.com/css?family=Oswald:400&text=NATE%20MAKES%20APPS%20FAST" rel="stylesheet">
```

This makes the font download required for my site a measly **1.4KB** in Chrome and Opera. Hell yeah.

But Nate, I want to do it all myself!

Yeah, I get it. Depending on Big Bad Google (or any 3rd-party provider) never makes you feel good. But, let's be realistic:

- Are you going to implement `unicode-range` optimization yourself? What if your designer changes fonts?
- Are you going to come up with 30+ varieties of the same font, like Google Fonts does, to serve the perfect one to each user agent?
- Are you going to strip the font-hinting from your font files to save an extra couple of KB?
- What if a new font technology comes out (like WOFF2 did) and even more speed becomes possible? Are you going to implement that yourself?
- Are you *absolutely sure* that there's no major benefit afforded by users having already downloaded your font on another site using Google Fonts?

There are some strange strategies out there that people use when trying to make WebFonts faster for themselves. There's a few that involve LocalStorage, though I don't see the point when Google Fonts uses the HTTP cache like a normal, respectable webservice. Inlining the fonts into your CSS with data-uri makes intuitive sense - you're eliminating a round-trip or two to Google - but the benefit rarely pans out when compared to the various other optimizations listed above that Google Fonts gets you *for free*. Overall, I think the tradeoff is clearly in Google's favor here.

Checklist for Your App

- **Stylesheets go before Javascript unless** . Unfortunately, Typekit only says to "put your embed code near the top of the head tag". If Typekit (or any other font-loading Javascript) is higher up in the `<head>` than your stylesheets, your users will be seeing a blank page *until* Typekit loads. That's not great.
- **If you have FOUC problems, either load fewer fonts or use webfonts.js**. Soon, we'll get the ability to control font fallback natively in the browser, but until then, you need to use [WebFontLoader](#). It may be worth *inlining* WebFontLoader (or its smaller cousin, [FontFaceObserver](#)) to eliminate a network round-trip.
- **If you can, use Google Fonts**. Google Fonts does a lot of optimizations you cannot realistically do yourself. These include stripping font-hinting, serving WOFF2 to capable browsers, and supporting `unicode-range` . In addition, you benefit from *other* sites using Google Fonts which may cause users to have already loaded the font you require!
- **Audit your WebFont usage**. Use Chrome DevTools to decipher what's going with your fonts. Use similar system fonts when text is too small to distinguish between fonts. WebFont downloads should almost always be less than 100KB.

Further Optimization

Here are some links for further reading on making WebFonts fast:

- [Ilya Grigorik, Optimizing WebFont Rendering Performance](#)
- [Adam Beres-Deak, Loading webfonts with high performance on responsive websites](#) Using LocalStorage to store and serve WebFonts. Try this one in your browser with Chrome Timeline open - it performs far worse than Google Fonts on first load.
- [Patrick Sexton, Webfont options and speed](#) Great overview of the multitude of

options available to you outside of Google Fonts.

- [Filament Group, Font Loading Revisited](#)

HTTP/2 and You

HTTP/2 is coming! No, wait, HTTP/2 is here! [After publication in Q1 of 2015](#), HTTP/2 is now an "official thing" in Web-land. As of writing (December 2015), [caniuse.com estimates about 70% of browsers globally can now support HTTP/2](#). That means I can use HTTP/2 in my Ruby application *today*, right? After all, Google says that [some pages can load up to 50% faster just by adding HTTP/2/SPDY support](#), it's magical web-speed pixie dust! Let's get it going!

Well, no. Not really. Ilya Grigorik has written an experimental HTTP/2 webserver in Ruby, but it's not compatible with Rack, and therefore not compatible with any Ruby web framework. While [@tenderlove](#) has done [some experiments with HTTP/2](#), Rack remains firmly stuck in an HTTP/1.1 world. [While it was discussed that this would change with Rack 2 and Rails 5](#), little actually changed. Until the situation changes at the Rack level, Rails and all other Ruby web frameworks are stuck with HTTP/1.1.

Part of the reason why progress has been slow here (other than, apparently, that [@tenderlove](#) is the only one that wants to work on this stuff) is that Rack is thoroughly designed for an HTTP/1.1 world. In a lot of ways, HTTP/2's architecture will probably mean that whatever solution we come up with will bear more resemblance to ActionCable than it does to Rack 1.0. Certain situations that are fairly simply in HTTP/1.1 land, such as multi-part POSTs, become far more complicated in HTTP/2 because of the protocol's evented nature.

Ilya Grigorik, Google's public web performance advocate, [has laid out 4 principles for the web architecture of the future](#). Unfortunately, Rack is incompatible with most of these principles:

- **Request and Response streaming should be the default.** While it isn't the default, Rack at least supports streaming responses (it has for a while, at least).
- **Connections to backend servers should be persistent.** I don't see anything in Rack that stops us from doing this at the moment.
- **Communication with backend servers should be message-oriented.** Here's one of the main hangups - Rack is designed around the request/response cycle. Client makes a request, server makes a response. While we have some limited functionality for server pushes (see [ActionController::Live::SSE](#)), communication in Rack is mostly designed around request/response, not arbitrary messages that can go in either direction.

- **Communication between clients and backends should be bi-directional.**

Another problem for Rack - it isn't really designed for pushes straight from the server without a corresponding request. Rack essentially assumes it has direct read/write access to a socket, but HTTP/2 complicates that considerably.

If you're paying attention, you'll realize these 4 principles sound a hell of a lot like WebSockets. HTTP/2, in a lot of ways, obviates Ruby developers' needs for WebSockets. [As I mentioned in my guide to ActionCable](#), WebSockets are a layer *below* HTTP, and one of the major barriers of WebSocket adoption for application developers will be that many of the things you're used to with HTTP (RESTful architecture, HTTP caching, redirection, etc) need to be *re-implemented* with WebSockets. Once HTTP/2 gets a JavaScript API for opening bi-directional streams to our Rails servers, the reasons for using WebSockets at all pretty much evaporate.

When these hurdles are surmounted, HTTP/2 could bring, potentially, great performance benefits to Ruby web applications.

HTTP/2 Changes That Benefit Rubyists

Here's a couple of things that will benefit almost every web application.

Header Compression

One of the major drawbacks of HTTP 1.1 is that headers cannot be compressed. Recall that a traditional HTTP request might look like this:

```
accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
accept-encoding:gzip, deflate, sdch
accept-language:en-US,en;q=0.8
cache-control:max-age=0
cookie:_ga=(tons of Base 64 encoded data)
upgrade-insecure-requests:1
user-agent:Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.73 Safari/537.36
```

Cookies, especially, can balloon the size of HTTP requests and responses.

Unfortunately, there is no provision in the HTTP 1.x specification for compressing these - unlike response bodies, which we can compress with things like `gzip`.

Headers can make up 800-1400KB of a request or response - multiply this to Web Scale and you're talking about a *lot* of bandwidth. HTTP/2 will reduce this *greatly* by compressing headers with something fancy called Huffman coding. You don't really need to understand how that works, just know this - HTTP/2 makes HTTP headers smaller by nearly 80%. And you, as an application author, won't need to do anything to take advantage of this benefit, because the compression/decompression will happen at lower levels (probably in Rack or some new layer directly below).

This compression will probably be one of the first HTTP2 features that Rails apps will be able to take advantage of, since header compression/decompression can happen at the load balancer or at the web server, before the request gets to Rack. You can take advantage of header compression today, for example, by placing your app behind Cloudflare's network, which provides HTTP/2 termination at their load balancers.

Multiplexing

Multiplexing is a fancy word for two-way communication. HTTP 1.x was a one-way street - you could only communicate in one direction at a time. This is sort of like a walkie-talkie - if one person is transmitting with a walkie-talkie, the person on the other walkie-talkie can't transmit until the first person lets off the "transmit" button.

On the server side, this means that we can send multiple responses to our client over a *single connection* at the *same time*. This is nice, because setting up a new connection is actually sort of expensive - it can take 100-500ms to resolve DNS, open a new TCP connection, and perhaps negotiate SSL.

Multiplexing will completely eliminate the need for domain sharding, a difficult-to-use HTTP 1.x optimization tactic where you spread requests across multiple domains to get around the browser's 6-connections-per-domain limit. Instead of each request we want to make in parallel needing a new connection, a client browser can request several resources across the same connection.

I mentioned domain sharding was fraught with peril - that's because it can cause network congestion. The entire reason the 6-connections-per-domain limit even exists is to limit how much data the server can spit back at the client at one time. By using domain sharding, we run the risk of *too much data* being streamed back to clients and causing packet loss, ultimately slowing down page loads. [Here's an awesome deconstruction of how domain sharding too much actually slowed down Etsy's page loads by 1.5 seconds.](#)

One area where Rails apps can take advantage of multiplexing today is by using an HTTP/2 compatible CDN for serving their assets.

Stream Prioritization

HTTP/2 allows clients to express preferences as to which requests should be fulfilled first. For example, browsers can optimize by asking for JS and CSS before images. They can *sort of* do this today by *delaying* requests for resources they don't want right away, but that's pretty jank and fraught with peril.

As an example, [here's an article about how stream prioritization sped up a site's initial paint times by almost 50%](#).

Again, your Ruby app can take advantage of this right now by using an HTTP/2 compatible CDN.

Latency Reduction

HTTP/2 will especially benefit users in high-latency environments like mobile networks or developing countries. [Twitter found that SPDY \(the predecessor to HTTP/2\) sped up requests in high-latency environments much more than in low-latency ones.](#)

Binary

HTTP/2 is a binary protocol. This means that, instead of plain text being sent across the wire, we're sending 1s and 0s. In short, this means HTTP/2 will be easier for implementers, because plain-text protocols are often more difficult to control for edge-cases. But for clients and servers, we should see slightly better bandwidth utilization.

Unfortunately, this means you won't be able to just `telnet` into an HTTP server anymore. To debug HTTP/2 connections, you're going to need to use a tool that will decode it for you, such as the browser's developer tools or something like WireShark.

One connection means one TLS handshake

One connection means TLS handshakes only need to happen once per domain, not once per connection (say, up to 6 TLS handshakes *for the same domain* if you want to download 6 resources from it in parallel).

Rails applications can experience the full benefit of this HTTP/2 feature today by being behind an HTTP/2 compatible web server or load balancer.

How Rails Apps Will Change with HTTP/2

All of the changes I've mentioned so far will generally benefit all Ruby web applications - but if you'll permit me for a minute, let's dive in to Rails as a specific example of your applications may have to change in the future to take full advantage of HTTP/2.

Primarily, HTTP/2 will almost completely upend the way Rails developers think about assets.

Concatenation is no more

In essence, all HTTP/2 does is make requests and responses cheaper. If requests and responses are cheap, however, suddenly the advantages of asset concatenation become less clear. HTTP/2 can transport a JS file in 10 parts pretty much as fast as it can transport that same file in 1 part - definitely not the case in HTTP/1.x.

In HTTP/1.x-world, we've done a lot of things to get around the fact that opening a new connection to download a sub-resource was expensive. Rails concatenated all of our Javascript and CSS into a single file. Some of us used frameworks like Compass to automatically sprite our images, turning many small .pngs into one.

But since HTTP/2 makes many-files just as cheap as one-file, that opens up a whole new world of advantages for Rails:

- Development mode will get waaaay faster. In development mode, we don't concatenate resources, meaning a single page often requires dozens of scripts and css files. HTTP/2 should make this just as fast as a single concatenated file in production.
- We can experiment with more granular HTTP caching schemes. For example, in today's Rails' world, if you change *a single line* in your (probably massive) application.js, the entire file will need to be re-downloaded by *all* of your clients. With HTTP/2, we'll be able to experiment with breaking our one-JS and one-CSS approach into several different files. Perhaps you'll split out high-churn files so that low-churn CSS won't be affected.
- We can amortize large amounts of CSS and JS over several page loads. In today's Rails world, you have to download *all* of the CSS and JS for the *entire application* on the first page load. With HTTP/2 and its cheap connections, we can experiment with breaking up JS and CSS on a more granular basis. One way to do it might be per-controller - you could have a single base.css file and then additional css files for each controller in the app. Browsers could download bits and pieces of your JS and CSS as they go along - this would effectively reduce homepage (or, I guess, first-page) load times while not imposing any additional costs when pages included

several CSS files.

Server push really makes things interesting

HTTP/2 introduces a really cool feature - server push. All this means is that servers can proactively *push* resources to a client that the client *hasn't specifically requested*. In HTTP/1.x-land, we couldn't do this - each response from the server had to be tied to a request.

Consider the following scenario:

1. Client asks for `index.html` from your Rails app.
2. Your Rails server generates and responds with `index.html`.
3. Client starts parsing `index.html`, realizes it needs `application.css` and asks your server for it.
4. Your Rails server responds with `application.css`.

With server push, that might look more like this:

1. Client asks for `index.html` from your Rails app.
2. Your Rails server generates and responds with `index.html`. While it's doing this, it realizes that `index.html` *also* needs `application.css`, and starts sending that down to the client as well.
3. Client can display your page without requesting any additional resources, because it already has them!

Super neat, huh? This will especially help in high-latency situations where network roundtrips take a long time.

Unfortunately, we can't tell whether or not the client has already cached the resource that we want to push to them. [There's an RFC currently under development to address this.](#)

Interestingly, I think some of this means we might need to serve different versions of pages, or at least change Rails' server behavior, based on whether or not the connection is HTTP/2 or not. Perhaps soon this will be automatically done by the framework, but who knows - nothing has been worked on here yet. It's not clear yet whether or not it will be your Rails application *or* the CDN pushing assets to the client.

How to Take Advantage of HTTP/2 Today

If you're curious about where we have to go next with hhh and what future interfaces might look like in Rails for taking advantage of HTTP/2, [I find that this Github thread is extremely illuminating](#).

For all the doom-and-gloom I just gave you about HTTP/2 still looking a ways off for Ruby web frameworks, take heart! There are ways to take advantage of HTTP/2 today *before* anything changes in Rack and Rails.

Move your assets to a HTTP/2 enabled CDN

An easy one for most Rails apps is to use a CDN that has HTTP/2 support. Cloudflare is probably the largest and most well-known.

There's no need to add a subdomain - simply directing traffic through Cloudflare should allow browsers to upgrade connections to HTTP/2 where available. The page you're reading right now is using Cloudflare to serve you with HTTP/2! Open up your developer tools to see what this looks like.

Use an HTTP/2 enabled proxy, like NGINX or h2o.

You should receive most of the benefits of HTTP/2 just by proxying your Rails application through an HTTP/2-capable server, such as NGINX.

For example, Phusion Passenger can be deployed as an NGINX module. NGINX, as of 1.9.5, supports HTTP/2. Simply configure NGINX for HTTP/2 as you would normally, and you should be able to see some of the benefits (such as header compression).

With this setup, however, you still won't be able to take advantage of server push (as that has to be done by your application) or the Websocket-like benefits of multiplexing.

Checklist for Your App

- **Use a CDN - preferably one that supports HTTP/2.** Using Rail's `asset_host` config setting makes this extremely simple.
- **If using NGINX, Apache, or a similar reverse proxy, configure it to use HTTP/2.** NGINX supports HTTP/2 in version 1.9.5 or later. Apache's `mod_http2` is available in Apache 2.4.17 and later.

Lab: HTTP2

This lab requires some extra files. To follow along, download the source code for the course and navigate to this lesson. The source code is available on GitHub (you received an invitation) or on Gumroad (in the ZIP archive).

This lab requires h2o, a webserver designed for HTTP/2. Install it via your favorite package manager:

```
brew install h2o
```

Provided is the actual homepage for the Ruby language. In this directory, in your shell, type:

```
$ ./serve_lab.sh
```

This will start two servers:

- <http://localhost:8000> is serving the Ruby homepage on HTTP 1.1.
- <http://localhost:8080> is serving the Ruby homepage with HTTP 2.0.

Exercise 1

Open both pages in the Network tab of your developer tools. Do you see any differences?

Spoiler:

The answer is, not really. You should notice some reduced network blocking at the beginning - since HTTP/2 opens only a single connection to our local h2o server, it is not limited by parallel connections like HTTP/1.x is (you'll see the queueing occur as a grey line in the network tab).

It's important to note that HTTP/2 is no performance panacea - certain sites will benefit, others (like ruby-lang.org) get minimal benefit.

Try to imagine a scenario, based on the previous lesson, of a page that would greatly benefit from an HTTP/2 enabled server. For example - try adding 20+ GIFs or images to the ruby-lang homepage. What happens?

Fast JavaScript for Rails Developers

JavaScript is increasingly becoming important on the Web. With single-page applications and JS frameworks all the rage, JavaScript as a percentage of page weight is ballooning. 250kb of JavaScript is considered "lightweight" for a Rails application now, and some frameworks will add *megabytes* of JavaScript to a page.

One of the troubles with JavaScript and page weight is that it's a double whammy - not only does the script need to be **downloaded** (this is where gzipped size matters), it needs to be **executed** (there is where non-gzipped, non-minified size matters). A small script, when gzipped, is not necessarily a fast one if it needs to execute a large amount of code before finishing.

This lesson is coming from the perspective of a non-Javascript-first developer, for other "non-Javascript-first" developers. If you've been using a JavaScript framework for 2+ years and it's your main language of choice, you can probably skip this lesson - you won't hear anything from me that you've never heard before. But for those of you who write more Ruby than you do JavaScript, read on.

Proper script tag usage

One of the most important things to realize about JavaScript and webpage performance is that the script tag itself is probably the most important determinant for JS performance.

Understand how scripts and stylesheets block the page load

As explained in some of the other front-end lessons in this course, `script` tags will always, in some fashion, block the page from loading. Here's the ways they can do that:

- **Network blocking** Like all sub-resources, `script` tags with a `src` attribute (often called "external scripts") will ask the browser's TCP connection pool for a connection (or open a new one) to `somedomain.com`. Browsers have limits on these pools - usually about six open connections per domain. Downloading an external script will take up one of these six slots, blocking other resources. You may see this concern (network blocking) brought up on old (2005-era) sites about web

performance, but it isn't much of a problem now. Browsers connection limits back then used to be just 2 connections per domain, so back then downloading a script meant you were almost certainly blocking some other important resource.

Nowadays, you have 6 parallel connections available so that's far less likely. In addition, as HTTP/2 rolls out, this concern goes completely out the window, as we can serve as many responses as we want over a *single* HTTP/2 connection to a domain.

- **Parser blocking** One of the first things the browser does is attempt to parse your HTML ("Parse HTML" in Chrome Timeline). As the parser parses the document, if it comes upon a script tag, it must *stop* and **block** until the script has downloaded (if it's external and not inlined) *and* executed. [Here's an example of parser blocking](#) - nothing appears until the script (with an artificial 2 second delay) has been downloaded and executed. Parser blocking only affects parts of the DOM that appear *after* the script tag in your HTML.

Of the two, parser blocking is far more important.

In addition, Javascript cannot execute until [the CSS Object Model \(CSSOM\)](#) has been constructed. This means that given the following DOM:

```
<link rel="stylesheet" media="all" href="/assets/application.css">
<script src="/assets/application.js"></script>
```

Assume `application.css` takes 2 seconds to download, and `application.js` takes 1 second to download but 3 seconds to execute. How long does it take to load the page? 4 seconds? 5 seconds?

[It takes 5 seconds - see for yourself](#). The critical rendering path looks like this: download the stylesheet and construct CSSOM (2 seconds), execute `application.js` (because it's already downloaded by this point). We can't execute `application.js` until `application.css` is downloaded and executed!

Don't script inject

Script injection is when an inline script tag inserts another, external script tag in the document. The most famous example is Google's Analytics tag:

```
<script>
(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
(i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.createElement(o),
m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)
})(window,document,'script','//www.google-analytics.com/analytics.js','ga');
ga('create', 'UA-XXXX-Y', 'auto');
ga('send', 'pageview');
</script>
```

Let's unpack it so that the script injection is clearer:

```
<script>
(function(window, document, strScript, url, variableName, scriptElement, firstSc
ript) {
    window['GoogleAnalyticsObject'] = variableName;
    window[variableName] = window[variableName] || function() {
        (window[variableName].q = window[variableName].q || []).push(arguments);
    };
    window[variableName].l = 1 * new Date();
    scriptElement = document.createElement(strScript),
    firstScript = document.getElementsByTagName(strScript)[0];
    scriptElement.async = 1;
    scriptElement.src = url;
    firstScript.parentNode.insertBefore(scriptElement, firstScript)
})(window, document, 'script', '//www.google-analytics.com/analytics.js', 'ga');
ga('create', 'UA-XXXX-Y');
ga('send', 'pageview');
</script>
```

All it's doing is inserting a script node with some particular properties, and then using the `ga` object to queue some functions up.

There's a problem however - in 2008, browsers started implementing these things called *preloaders*. Preloaders speculatively scan the document, ahead and separate from the main parser, to look for things that might need to be downloaded. That way, when the parser finally gets to the DOM node, the preloader can just hand that resource to the parser and say "hey, already downloaded that for you."

Here's an example in action. Based on what you already know, and without a preloader, you would expect a page that has markup like this:

```
<link rel="stylesheet" media="all" href="/assets/application.css">
<script src="/assets/application.js"></script>
```

...to take 7 seconds to load if `application.css` takes 5 seconds to download and `application.js` takes 2 seconds. [With the preloader, however, it only takes 5](#). The reason is the preloader - while waiting for the stylesheets to download, your browser also downloads `application.js`.

So what does this have to do with script injection? Script injection prevents the preloader from doing its job. Preloaders are dumb - they can't execute Javascript. They basically are fancy regexes looking for "src" attributes and "href". They won't execute your inlined Javascript. [Open your Network tab when you check out this example](#). At first glance, it looks like our previous example - a 5 second page load. But investigation with the Network tab shows that the script doesn't start downloading until *after* the CSS has downloaded - 5 seconds later than our example with a regular script tag!

Don't use script injection - if you have to, you can deconstruct 3rd-party vendors that insist that you do and turn them into regular script tags. [Here's an example with Google Analytics](#).

You may be wondering - but if my stylesheet downloads quickly, I'm still blocking DOM construction! To which I reply, absolutely - this is why we have `async` !

Async defer all the things

As mentioned in other lessons, the `async` attribute is really as close as you can get to magical web performance pixie dust. The `async` attribute **removes a script tag from blocking the DOM parser** and executes the script only after it has finished downloading. This has the added advantage of speeding up the `DOMContentLoaded` event, which won't fire until the DOM parser has completed parsing the page.

[As an example, see this page in your Network tab](#). `DOMContentLoaded` fires immediately. Nearly all major websites, such as Github and Basecamp, use some combination of `async` and `defer` attributes.

`async` and its cousin, `defer` are slightly different in two important ways:

- `async` executes the script as soon as it is downloaded. `defer` executes it *after* `DOMContentLoaded` fires. If an `async` script finished downloading *before* `DOMContentLoaded`, it will be executed immediately, delaying `DOMContentLoaded` and any event handlers attached to that (like `$(document).ready()`).
- `defer` is supposed to guarantee execution order. `async` does not. If two script tags with `defer` appear in the document, the browser is supposed to execute them in order after `DOMContentLoaded`. Unfortunately, this behavior is buggy - don't rely on

it.

What this means is we can make the following guideline for using `async` and `defer` :

- **Use `defer` for advertisements and other JavaScript that can wait to execute.**
- **Use `async` for Javascript that should be executed as soon as possible, like analytics or user tracking.** When using `async` , you should also use a `defer` attribute. IE8 and IE9 do not support `async` , but support `defer` . These browsers will ignore the `async` attribute and execute the `defer` instruction. All other browsers will perform the `async` behavior and ignore `defer`.
- **Use neither for Javascript that is required for the page to render properly.** If your Javascript is obtrusive, you cannot use either attribute. An example might be a Typekit (the webfont provider) script tag - if you add an `async` attribute to Typekit's script tag, your page will suffer a flash of unstyled content, as your fonts will load *after* the text on your page has already been rendered. Another example might be a 3rd-party tool like Optimizely which changes page content after it has loaded. Adding `async` to an Optimizely script tag would also cause a FOUC.

In addition, you won't be able to use either attribute if your JavaScript must load in a particular order. To get around this, combine JavaScripts that must load in a particular order (for example, JQuery and `jquery_ujs`) into a single file.

Scripts at the bottom is mostly outdated and a workaround

"Put all JavaScript at the bottom of the document before the closing body tag" is some tried-and-true performance advice, and it isn't wrong. Check out this example - [moving the script tag to the top would delay the display of the page content by 2 seconds \(try it!\)](#).

However, `defer` gives us the same behavior with the advantage of not blocking `DOMContentLoaded` !

Scripts at the bottom has another huge issue - it doesn't work with Turbolinks. Turbolinks assumes that all JavaScript tags live in the HEAD - any scripts in the `body` are executed on each page change. Using `async` and `defer` allows us to get better performance than "scripts at the bottom" anyway, so that's no loss!

Combine scripts everywhere possible (HTTP/1.x)

In the world of HTTP/1.x, downloading many resources is costly and could trigger network blocking. TechCrunch.com, as an example, downloads dozens and dozens of scripts of varying sizes.

Javascript, like any other asset, should be concatenated to maximize performance in the world of HTTP 1.x. In HTTP/2, this isn't required - the browser can intelligently manage bandwidth such that the performance difference between downloading 10 scripts from a domain and 1 script that's 10 times as big is negligible.

There's no excuse for serving multiple JavaScripts from the same domain over an HTTP/1.x connection - concatenate!

Profiling

Profiling is important part of your performance workflow . However, when it comes to JavaScript, I recommend just sticking with Chrome Timeline. The flamegraph view especially is a more-than-adequate profiler for most applications that use JavaScript lightly to moderately.

Common mistakes

And finally, here's some common performance mistakes Rails developers make when working with JavaScript.

Cache DOM lookups

Don't lookup things twice - JQuery makes this "too easy" to mess up, unfortunately.

```
$('#some .complicated .selector').addClass('whoops');  
$('#some .complicated .selector').removeClass('ouch');
```

Especially inside of loops (scroll events!) this can be a real performance drag. Cache any DOM lookup that you call more than once:

```
var $thing = $('#some .complicated .selector')  
$thing.addClass('whoops');  
$thing.removeClass('ouch');
```


Minimize DOM size

Want to know how many DOM elements are on a page? Paste this in the console:

```
document.getElementsByTagName('*').length
```

A simple page is a fast page. Less DOM elements (and less nesting) means selectors perform faster. How many DOM elements is too many? Here's some famous sites and their total element counts, as of January 2016:

| Site | DOM Elements |
|---------------------|--------------|
| google.com | 439 |
| yahoo.com | 1728 |
| reddit.com | 2133 |
| twitter.com feed | 4727 |
| businessinsider.com | 2449 |
| forbes.com | 2072 |

Either use a SPA framework or Turbolinks

Unless you *aren't using JavaScript hardly at all* (let's say less than a 5kB download for your entire application), you should be using either a "single page application" framework like Ember or Turbolinks. Why?

Setting up a webpage is hard work. One thing most Javascript frameworks get *right* is they don't throw out the entire page when the user navigates. They just change the elements that need to be changed (think React), update the URL in the navigation bar, and carry on. A regular page navigation throws away an entire DOM, CSSOM, and Javascript VM. Most JavaScript frameworks *don't do this*, which makes them fast on navigation.

However, neither does Turbolinks. Turbolinks, like JavaScript frameworks, *keeps* the page around. It just replaces the `body` element with a new `body` element that it receives from the server via AJAX. For more about Turbolinks, see the lesson.

Either of these approaches can reduce navigation times between pages on your site. I prefer the progressive-enhancement approach of Turbolinks to the JavaScript-heavy approaches of Ember or React. But just use one of them - eliminate the work of setting up a new DOM and re-executing all of your JavaScript on every page load.

Understanding repaint and reflow

Reflow is the name of the web browser process for re-calculating the positions and geometries of elements in the document, for the purpose of re-rendering part or all of the document. -[Google](#)

A repaint occurs when changes are made to an elements skin that changes visibility, but do not affect its layout. Examples of this include outline, visibility, or background color. -[Nicole Sullivan](#)

Often, the browser will need to *reflow* or *repaint* all or portions of the document to deal with changes to the CSS Object Model. From a performance perspective, reflows (sometimes called re-laying out) are considerably more important and taxing than a repaint.

The most common causes for reflow are:

- Adding stylesheets. The second stylesheet tag in a document, for example, causes a reflow as the new styles are added.
- Manipulating classes of elements and changing `style` attributes.
- Calculating certain numbers, like `elem.offsetLeft`.

[Here's an exhaustive list of things that can cause reflow.](#)

What can you do about it?

First, open up Chrome Timeline. The tools available are excellent for detecting reflows (Chrome Timeline calls it "layout thrash" and marks potential incidents with a red flag in the timeline).

As mentioned above, a simple DOM is a fast DOM. Unnecessarily deep DOMs with dozens of levels will cause reflows to be slower, as reflows must propagate down the DOM tree. When changing classes on an element, do it as far down in the DOM tree as possible - the fewer the child elements, the faster it will be.

[For more about reflow, Paul Irish's gist is the canonical source.](#)

`$(kitchen_sink).ready()`

Rails developers have this terrible habit of dropping a *million* functions into

`$(document).ready()`. As these pile up, page load slows to a crawl as dozens and dozens of JavaScript functions must be executed every time the user navigates to a new page. God forbid that you're not using a single-page-app framework or Turbolinks, because attaching and executing all of these event handlers will happen on every single page!

Using `async` and `defer` helps here. But far better is to use event delegators smartly:

```
$("#myID").on("click", function() {...});
$(document).on("click", "#myID", function() {...});
```

The first function requires the DOM to be ready - otherwise if `$("#myID")` hasn't been parsed yet, the event handler will never attach. The second approach, however, simply *delegates* noticing those events to the document itself. This can be executed at any time (if attaching to the document), and does not have to wait for `DOMContentLoaded`. In addition, if elements are added or removed, you do not have to re-attach their event handlers.

You may notice that, in addition, the first method is not `async` friendly. Consider that line being included in an `async` script tag - what happens if the script executes *before* the DOM is ready?

In this way, event delegation is far superior in most cases to attaching event handlers directly.

Checklist for Your App

- **Use Turbolinks or a single-page-app Javascript framework.** If you're on the "JavaScript frameworks are great!" gravy train, great - keep using React or Angular or whatever else you guys think is cool this week (wink!). However, if you're *not*, you should be using Turbolinks. There's just too much work to be done when navigating pages - throwing away the entire DOM is wasteful as events must be re-delegated and handlers reattached, Javascript VMs built and DOMs/CSSOMs reconstructed on every page load.
- **Most pages should have no more than a few thousand DOM elements.** If a single page in your application has more than ~5,000 DOM elements, your selectors are going to be adversely affected and start to slow down. To count the number of elements on a page, use `document.getElementsByTagName('*').length`.
- **Look for layout thrash with Chrome Timeline.** Load your pages with Chrome Timeline and look for the tiny red flags that denote layout thrashing.

HTTP Caching

The fastest HTTP request is the one that is never made

- *Ancient programmer proverb*

The network will always be slow. For all the advances made in mobile and home bandwidth over the years, they will never eliminate one of the most fundamental causes of slow webpage loads - network latency. No, really. We can never completely solve this problem. At least, not without an Einstein-level breakthrough in our understanding of physics.

A packet of information will always be limited to the speed of light. It will *always* take at *least* 13 milliseconds for a client in New York to send a packet to a server in Los Angeles. Usually it will take much longer - *even fiber optic cable can only transmit photons at 60-70% of light's maximum speed*. The speed of light is the fundamental speed limit of the universe - and for our HTTP requests.

Even in a universe where we could invent a super-fiber-optic-cable that could transmit digital signals via light at 100% efficiency, we still have problems higher up the stack. Consider the following:

- Opening a new TCP connection to a new domain can involve as many as 4 or 5 network-round trips to resolve DNS, negotiate SSL, and complete a TCP three-way-handshake.
- Latency is often far worse than ideal due to network conditions and topography. Often, in developing countries, the route a packet takes from client to server is roundabout and confusing due to poor infrastructure (or even satellite internet connections).
- Mobile is worse. Even on modern 4G networks, we're talking ~200-400 milliseconds, and on 3G round-trip times can reach a full second.

This is why HTTP caching is so important. It will always be significantly faster to *not make a request at all* than it will be to make one.

In addition to speeding up end-user experiences, HTTP caching reduces load on your own servers. Proper HTTP caching can shunt a huge amount of bandwidth to 3rd-party CDNs (by making content cacheable) or eliminate it entirely. Every request served from the client cache (or an intermediate cache) is one that you didn't have to serve yourself.

For Ruby developers, HTTP caching is most useful in these scenarios:

- Static assets. This is the most common use case, and the one you're probably familiar with. In a perfect world, your Rails application should have almost nothing to do with serving your static assets. HTTP caching helps us to do that.
- JSON APIs and AJAX endpoints. Any endpoint in your application that serves JSON (or XML or any other non-HTML resource) is a great candidate for HTTP caching.

Why not cache HTML documents? Unfortunately, every Rails application protects you from Cross-Site-Request-Forgery using a token included in the `head` tag of the HTML response. This token is different for every user session and every page load. You've almost certainly seen it before:

```
<meta name="csrf-param" content="authenticity_token" />
<meta name="csrf-token" content="Xc0vf6L7hgbI6zRPBpNhgqrLhX7sMQMRltiFEjryce81q09yU
cPy1LPn07YKx6rzLyLQ/F/pSCpSPc8uXXRdwg==" />
```

This makes HTML documents in Rails pretty much uncacheable. This token changes on every page load, and HTTP caching only works with resources that remain completely unchanged. There are some ways around this (edge-side server includes, getting the CSRF token with AJAX) but none of them are really satisfactory or recommended.

We don't usually have this problem with JSON APIs - they typically don't have CSRF problems, usually because they lack any concept of a user session. If you don't have CSRF tokens on your HTML pages, you can also use HTTP caching for those documents.

Anatomy of a Cache Header

HTTP caching is implemented through HTTP headers. Here's the headers for a typical `application.js` served by a Rails application:


```

HTTP/1.1 200 OK
Server: Cowboy
Date: Mon, 18 Jan 2016 18:56:47 GMT
Connection: keep-alive
Last-Modified: Thu, 14 Jan 2016 18:09:41 GMT
Content-Type: application/javascript
Cache-Control: public, max-age=31536000
Vary: Accept-Encoding
Content-Length: 113134
Via: 1.1 vegur

```

The relevant cache header here, and the one which implements about 80% of HTTP caching behavior, is `Cache-Control`. To break down this particular header:

- `public` specifies that this resource can be cached by edge or "intermediate" caches. Anything that isn't the end-user's browser is generally considered an "edge" cache. Edge caches include things like CDNs or other intermediate caches like [Varnish](#)).
- `max-age` tells the cache (either the browser or an intermediate cache, like your CDN) how long it may store the resource. This value is expressed in *seconds from now* - a `max-age` of 60 expires a minute from the time of the request. In this case, the resource expires 1 year from now.

These are, actually, Rails' default cache headers for assets. They're great defaults for Rails - but there's a wide world out there in HTTP caching, and Rails use case here (unique filenames with a changing digest, static javascript and stylesheets) will not fit every situation.

How HTTP Caches Work: Entity Tagging (ETags) and Revalidation

Imagine our browser downloads the `application.js` resource above. Based on the cache-control headers, it knows it can cache the resource and not request it again until a year from today. Any subsequent requests for the resource will be served from the local cache - no network requests required!

Let's say our web browser, Rip van Winkle of the Web, sleeps for the next year and doesn't visit any other webpages in that year. This is important - browser caches have a limited size, and work like any other LRU cache. Entries which haven't recently been used will be evicted. So after that year is up, Rip Van Winkle goes to `yourdomain.com` again.

The browser notices it needs the `application.js` subresource, but its cached version is expired. The browser will send a new request, but it will contain an additional header:

`If-Modified-Since: Thu, 14 Jan 2016 18:09:41 GMT`. The origin server (or intermediate CDN cache, if one exists) will look at this header and compare it to the requested resource's `Last-Modified` date. If the resource has been modified, the origin server will respond with the full resource. If it hasn't, the origin server responds with a `304 Not Modified` response, saving the browser from downloading the entire resource again. Rip van Winkle can use his cached copy of `applicaton.js` for another year!

There's an alternative way to revalidate resources too - Entity Tags, or ETags. ETags are typically just short hashes or digests of the resource. If the resource changes, the ETag changes - just like the digest attached to a Rails asset.

Instead of sending an `If-Modified-Since` header, a browser will send an `If-None-Match: some-digest12345` header when revalidating an ETagged resource. If the value of the `If-None-Match` header matches the origin server's ETag for that resource, a `304 Not Modified` response is served. ETags are useful for resources where a "last modified" date is unclear, like a dynamic HTML or JSON resource. Imagine a homepage - when was it "last modified"? It's usually clearly to just create a digest for the resource and use it as an ETag.

By default, Rails uses last-modified dates for static assets rather than ETags.

Unfortunately, it appears that ETags have [several problems](#) in Rails - watch the linked issues for progress in these areas. For 90% of applications, you'll want to work with setting the proper `Last-Modified` header.

Cache-Control

There are several components to a `Cache-Control` header, called *directives*. Here's an explanation of the useful ones and when you might want to use each. Most of them (except `public` and `private`) can be combined with each other.

- `no-store`. As the original HTTP spec says: "The purpose of the no-store directive is to prevent the inadvertent release or retention of sensitive information (for example, on backup tapes)". `no-store` prevents any client from storing the response at all. You might want to include this directive for things like sensitive responses from JSON APIs (payment data, order or customer information, etc). I can't think of a good reason to use this on a static asset. This prevents *all* caching of a resource *anywhere*.
- `no-cache`. Confusingly, while *you* may think of "caching" and "storing" being

basically the same thing, HTTP caching doesn't think this way. A resource with the `no-cache` directive *will* be stored on intermediate caches or CDNs, and *will* be stored in the users browser. However, caches will *always revalidate* the cached response with the server. This is basically the same as setting a `max-age=0`, as noted below. This would only be really useful in the case of a resource with a large filesize (say, north of 100kB) that must be revalidated but is often unchanged. `no-cache`, although it still incurs a round-trip between the server and the cache, can prevent downloads from occurring if revalidation is successful.

- `public` and `private`. `private` prevents intermediate caches, like CDNs, from caching the response. Use it, perhaps along with when caching a resource on your CDN would raise a privacy concern. `public`, which is basically the default behavior, just specifies that a resource may be cached by any cache.
- `max-age`. The value, in seconds from now, when the cache should consider this resource expired and revalidate it with the origin server. Pretty obvious - set this based on your needs. You can set this as far in the future as you want.
- `no-transform`. Straight from the spec: "Implementors of intermediate caches (proxies) have found it useful to convert the media type of certain entity bodies. A non-transparent proxy might, for example, convert between image formats in order to save cache space or to reduce the amount of traffic on a slow link." Lots of CDNs today will modify content, like images, to try to speed up requests. If for some reason you would want to stop your intermediate caches from doing this, add `no-transform`.
- `must-revalidate`. You might think this is the same `no-cache` - well, it isn't. `must-revalidate` only refers to revalidation behavior *after* the resource has expired (say, after its `max-age` passes by). The reason this header exists is that caches *may* choose to ignore `max-age` or replace it with their own value. `must-revalidate` forces the cache to obey the expiration times in the Cache-Control header.

There's an extremely good flowchart in [Google's guide to HTTP cache headers](#) available [here](#).

HTTP Caching and Assets

Although this is discussed further in the lesson on CDNs, my recommended "static file serving" setup looks like this:

- Your Rails process should turn on static file serving OR, if using Apache or NGINX, serve static files from the `/public` directory.

- Use a CDN that uses your Rails server as its origin. On the first request, the CDN fetches the asset from your Rails process. From then on, your CDN will serve the asset until it expires.

This results in your Rails server process serving each asset once, and from then on each asset will be served by the CDN. It imposes basically no load on our production servers, and eliminates complicated and costly uploading to a third party service, like S3. Of course, if you have many assets which are either dynamically generated, user uploaded, or extremely large, you may want to upload them somewhere else anyway.

This setup requires proper Cache-Control headers, otherwise your CDN may not cache your static assets at all. When using Rails and the Asset Pipeline, setting proper Cache-Control headers is rather simple.

```
# Rails 4
config.serve_static_assets = true
config.static_cache_control = "public, max-age=31536000"

# Rails 5
config.public_file_server.enabled = ENV['RAILS_SERVE_STATIC_FILES'].present?
config.public_file_server.headers = {
  'Cache-Control' => "public, max-age=31536000",
  # in Rails 5, we can also add other headers here for static assets if you like
  'Server' => 'Rails Asset Server'
}
config.asset_host = "mycdn.com"
```

That's it - simple as that.

HTTP Caching and JSON APIs

As stated above, HTTP caching for HTML documents is pretty much useless if you use CSRF protection (which you should). If for some reason your app doesn't use CSRF tokens embedded in the HTML, you may be able to apply this section in your application too. APIs typically don't have sessions, so this section is primarily aimed at them.

In Rails, you can get a general idea of the HTTP caching API by reading the [ActionController::ConditionalGet](#) module. This module is included in `ActionController::Base` by default, so all of its methods are available on your controllers. In addition, [Heroku maintains an excellent guide on HTTP caching in Rails](#).

However, `ActionController::ConditionalGet` is just sugar on top of setting `cache-control` and other headers, so rather than discuss those sugar methods specifically, I'll discuss the use of caching headers for API resources. Read the linked guides to figure out how to implement these guidelines with Rails.

HTTP caching to prevent database queries

A great use of HTTP caching on an API is preventing unnecessary database queries. If the client has already requested and cached the resource, and needs to revalidate it, we *shouldn't* trigger a bunch of database queries if it's obvious the object hasn't been updated.

Consider a Twitter-like application. A `User` resource has a `Timeline` resource, and to get a list of `Microposts` we ask for the users' `Timeline` . If the `Timeline` hasn't changed since the client last requested it, we should serve a 304 Not Modified.

Here's some Ruby psuedo-code with how you might handle this:

```
@timeline = user.timeline
if request.headers['If-Modified-Since'] == @timeline.microposts.maximum(:updated_at)
  # trigger 1 simple, fast SQL query to get the
  # respond with 304 Not Modified
else
  # respond with a timeline, triggering the full SQL query
end
```

Rails makes this flow considerably easier with the `fresh_when` method:

```
def show
  @timeline = user.timeline
  fresh_when last_modified: @timeline.microposts.maximum(:updated_at)
end
```

Recall that this all works because ActiveRecord lazily loads the actual SQL query - `user.timeline` doesn't actually execute a SQL query, because we haven't used the results yet, but `user.timeline[0]` will.

Use a request's provided `ETag` and `If-Modified-Since` headers to determine if any queries need to be re-executed - otherwise, try to serve a 304 Not Modified.

Public and private - controlling cache copies

When implementing HTTP caching for an API, be sure you're conscious of where these resources are being cached. As noted in the discussion of `public` vs `private` above, anything marked `public` will be stored on CDNs. This can be a good thing or a bad thing - it depends on the resource. **Note that Rails marks all HTTP cacheable resource in a controller as `private` by default.** If you want an API response to be cached by an intermediate CDN, best to mark it `public` explicitly.

If you're serving something that's private and shouldn't be cached at all, either by the browser or intermediate CDNs, use the `no-store` directive.

Make sure the clients have response caches!

Most Ruby HTTP clients, like `rest-client` or `Net::HTTP`, do not have caches. As of this writing, January 2016, only the following Ruby HTTP clients have response caches:

- Faraday, via [faraday-http-cache](#)
- [Typhoeus](#), which makes sense, since it's just a curl wrapper.

If you're running an API server, be sure that your clients actually have response caches too! If you're using your Ruby application as a backend for a JavaScript-powered browser application, you're already set - AJAX requests will just use the browser's cache as normal.

One last tip: minimizing churn

Often, especially in Rails world, the pattern for Ruby web applications is to serve a single `application.js` and/or `application.css` file. This is good for several reasons - faster downloading over HTTP/1.1 where parallel downloads can block, and concatenating into a single file gives bigger gains from compression algorithms like gzip.

There is *one* good case though, where I'd recommend deviating from this "one file" approach - when components of your `application.js` or `application.css` churn significantly differently from the rest. Do you have one part of your Javascript that's constantly changing? Or is your application-specific CSS changing constantly, but your vendored CSS (like Bootstrap, perhaps) relatively stagnant?

In this case, we can make greater use of HTTP caching by separating assets into heavily churning and lightly churning files.

Let's take the Javascript example. Let's say we're working on a JS-heavy app that uses React. Our `application.js` file may be changing several times *per day* because we're pushing so many changes to our application code all the time. However, the project hardly ever updates React and its other vendored Javascript libraries - more like once a month than thrice a day.

In this case, it would be far better to split the single `application.js` into two files - a `libraries.js` where all of our vendored scripts live, and an `application.js` where all of our app-specific code lives. This way, `libraries.js` will more frequently be able to be served from the HTTP cache, reducing page weight for our returning users. I suspect this could be a common optimization for JS-heavy (or even CSS-heavy!) applications with lots of repeat visitors.

Be careful when splitting into more than 2 files or so - eventually, you'll start to cause network blocking if these assets are served over HTTP/1.1. With HTTP/1.1, we have to open new connections to download assets simultaneously, *and* we can only have six of these connections open at a single time.

If you're serving your assets over HTTP/2 (over Cloudflare, for example), you have a lot more leeway, as there is no parallel connection limit (all assets are downloaded over 1 connection).

When experimenting with splitting assets for improved caching, be sure to test the performance impact in DevTools.

Checklist for Your App

- **Experiment with splitting your `application.js/application.css` into 2 or 3 files.** Balance cacheability with the impact to initial page download time. Consider splitting files based on churn (for example, one file containing all the libraries and one containing all of your application code). If you're using an HTTP/2-enabled CDN for hosting your static assets, you can try splitting them even further.
- **Double-check to make sure your site has sane cache control headers set.** Use Chrome's Developer Tools Network tab to see the cache control headers for all responses - it's an addable column.
- **If running an API, ensure that clients have response caches.** Most Ruby HTTP libraries do not have response caches and will ignore any caching headers your API may be using. Faraday and Typhoeus are the only Ruby libraries that, as of writing (Feb 2016), have response caches.

- **Make sure any user data is marked with `Cache-Control: private`.** In extreme cases, like passwords or other secure data, you may wish to use a `no-store` header to prevent it from being stored in any circumstance.
- **If a controller endpoint receives many requests for infrequently changed data, use Rails' built-in HTTP caching methods.** Unfortunately, Rails' CSRF protection makes caching HTML documents almost impossible. If you are not using CSRF protection (for example, a sessionless API), consider using HTTP caching in your controllers to minimize work. See [ActionController::ConditionalGet](#)

Lab: HTTP Caching

Exercise 1

Describe the appropriate HTTP headers for the following resources:

- A static asset, such as Javascript or CSS, with a digested file name ("application-4cc81...").
- A static asset *without* a digested file name ("index.js")
- A JSON response with some private user data, like contact information.
- A JSON response with some bank account and credit card numbers.

Solution

- `Cache-Control: max-age=315360000` , or 1 year.
- `Cache-Control: max-age=300` , or a similarly short time, with a strong ETag header, such as: `ETag: "686897696a7c876b7e"`
- `Cache-Control: private` , with a max-age as appropriate.
- `Cache-Control: no-store`

Exercise 2

This lab requires some extra files. To follow along, download the source code for the course and navigate to this lesson. The source code is available on [GitHub](#) (you received an invitation) or on [Gumroad](#) (in the ZIP archive).

The included Rails application in the `lab` folder (see `app.ru`) gets weather reports. Using HTTP caching, instruct browsers to cache their results for up 30 minutes. Hint: the weather JSON contains some information that can be used to set some dates in your cache headers!

Solution

See `app_solution.ru` .

Module 3: Ruby Optimization

This module is all about the nitty-gritty of Ruby performance in our web application. This area offers a lot of room for improvement for almost any Rails application.

The most important lesson in this module is on **caching**. Every Rails application should aggressively use caching, and few do. That lesson discusses why caching is so important, breaks down Russian Doll caching, and provides a benchmark and overview of the cache backend options available to you.

Bloat Management

In production, most Ruby applications are memory-constrained. Popular hosts and deployment targets, like Heroku, DigitalOcean, or even Amazon Web Services force Ruby applications to contend with small amounts of available RAM. Heroku's default dyno size is still just 512MB.

512MB feels like peanuts nowadays. I've seen Rails applications where a *single instance* of the app takes up more than 512MB. As covered in my lesson on slimming down your Rails application, most Rails applications take up ~150-200MB upon startup. After a few hours of running and "burn-in", that can easily double, and maybe even triple.

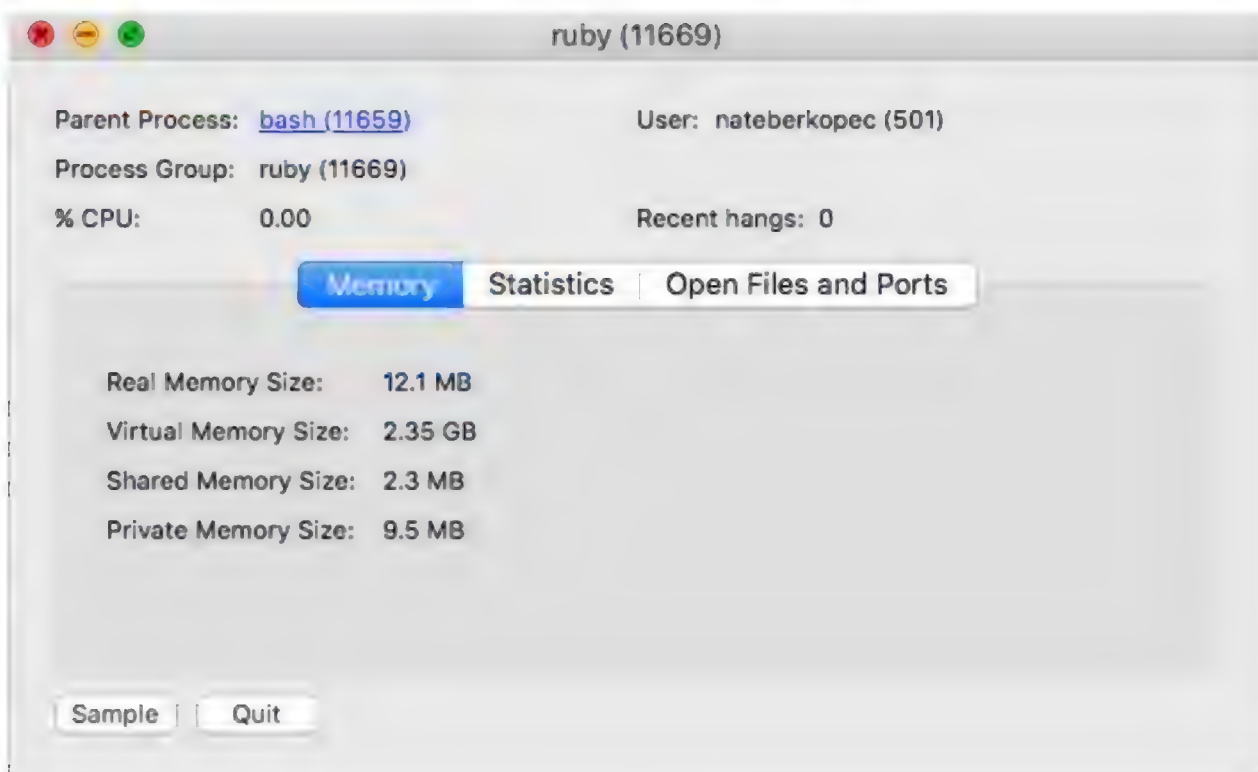
This means that the memory usage of our applications is one of our biggest opportunities for improving scalability. In most cases, decreasing memory usage directly leads to more application instances for the same given amount of production resources.

This article is written for Unix systems - if you're deploying Ruby on Windows, you're on your own here. The way Windows manages memory is just too different from Unix.

How do we measure the memory usage of a Ruby application?

The question of "how much memory is this Ruby application using?" is surprisingly difficult to answer.

First, you have the nebulous word itself: "memory"? What exactly do we mean by "memory"?



Definitions of memory usage

It turns out that memory is much more complicated than a given bit of memory being "used" or "free". There are several types of memory (or, more accurately, ways of *describing memory usage*) on a modern system:

- **Shared Memory** This is memory accessible by any other process on the system. Most read-only, non-writable memory is simply marked "shared" because there isn't any reason not to share it. Shared memory is memory used by your application.
- **Private Memory** Memory that a process uses for itself and its forked child processes. This is memory that is only being used by your application. This memory can be shared with forked child processes, e.g. Puma workers.
- **Real Memory** The amount of physical memory that the operating system has given your process. It should, roughly, correspond to the sum of private and shared memory.
- **Virtual Memory** This is just the amount of memory accessible to your process - it has nothing to do with how much it's actually using. When a virtual memory address is accessed, the operating system connects the virtual address to a real one. Since the access is virtualized, that means the actual memory can be anywhere - in RAM, or in the hard disk, or somewhere else entirely. Note in my screenshot that the virtual memory for a fresh `irb` session is several gigabytes - `irb` doesn't actually need that much to run!

- **Swap Memory** When RAM is full, your operating system will start to use the hard disk as if it was RAM. This is called swapping. Generally, this is a *bad thing*, because accessing data from a hard disk is quite a bit slower than accessing it from RAM. However, operating systems *may* start using swap memory *before* the RAM fills up - at any time, the OS may move memory into the swap space to free up more memory for other processes. The key thing is to know whether or not the swap memory is actually being used. If our swap usage is large, but we're not actually reading or writing anything from it, we don't have a problem - our OS is just being efficient in where it allocates its memory!

How do we measure it?

Now that we're all caught up on terms, how do we measure memory usage?

- **Resident Set Size (RSS)** In a simple scenario, RSS is equal to the amount of physical RAM pages (thus, real memory) used by the process. However, as we just learned, some memory can be shared between processes - so RSS isn't really equal to "amount of memory that would be freed if I kill this process". If two processes have 5MB of private memory and share 5MB of memory between them, they both have an RSS of 10MB, but killing one of those processes would only free up 5MB of memory. However, since memory can be swapped (and thus becomes part of the "virtual" memory used by your process), it isn't a fully accurate representation of the "total" memory usage of your process. Memory leaks can sometimes have constant RSS usage for this reason - as the memory slowly leaks, it is moved to swap (becoming part of the virtual memory space), and is never accessed again (removing it from the resident set).
- **Proportional Set Size PSS** PSS is pretty much the same as RSS, except in the way they deal with memory pages. RSS calculates memory usage in physical RAM by simply counting the number of memory pages used. PSS takes the number of pages, but only assigns $1/N$ of each page to a process where "N" is the number of processes currently using a particular memory page. For this reason, PSS is usually smaller than RSS. It's also, generally speaking, a more accurate representation of the amount of memory usage. However, since PSS needs to know how many processes are accessing a particular memory page, it is *impossible* to get PSS accurately from inside of a container (Docker, Heroku, AWS, other virtual server environments). It is possible to determine these numbers accurately from *outside* of the container.

In practical terms, here are some tools I use when debugging memory usage of a Ruby application:

- `ps` - I use `ps` with its `-o` option to get some specific information about memory usage. `ps | grep '[r]uby' | awk '{print $1}' | xargs ps -o rss,vsz,nswap` grabs the PID of all ruby processes, then feeds it back to `ps 's -o` option to report the resident set size, virtual size, and number of swaps in and out. I look at RSS for a general measure of much memory this process uses right now, and I watch virtual size and number of swaps to see if they're growing over time. Growing virtual size indicates possible leaking, and a growing number of swaps means that my system is under memory pressure and frequently swapping this process' memory to disk. Note that increasing *real* memory doesn't necessarily indicate a leak - increasing real memory is just the natural consequence of virtualized memory becoming real memory over time.
- `get_process_mem` is a great tool for measuring RSS *inside* of a Ruby process - useful if you want to use that number to do other things!

Great - so now you know how to accurately get the picture of how much memory your Ruby process is using. Now that you *know* your memory usage, how can we start reducing it?

Reducing Memory Bloat

There are a lot of ways to reduce memory bloat - here's a few of the ones I've found useful.

Beware Big Allocations

Ruby uses memory in a funny way - it tends not to release memory *back* to the operating system after it is used.

If you create an enormous 500MB array, Ruby asks `malloc` to allocate memory. The amount it needs depends on how much memory is already allocated for the heap - let's say it has 100MB, so Ruby will ask for an additional 400MB to be allocated for the heap. All well and good - your 500MB is allocated. Now let's say that array dies off and gets garbage collected. What's the memory usage of your Ruby process now? 100MB?

Nope. It's still at 500MB.

Don't believe me? Try this one in any `irb` session:

```
Array.new(1_000_000) { "string" }
```

This will create an array of 1 million string objects. Check your `irb` process' memory usage with `ps` - mine ballooned to 350MB. Now, manually trigger garbage collection:

```
GC.start
```

You might save about ~10% of your process' memory usage, but it's still a huge number!

Once you've allocated memory to Ruby, Ruby gives it back to the operating system slowly. You can verify this by checking `GC.stat[:heap_free_slots]` - the number will be huge!

This means that momentary memory pressure, whether from many small objects or just a few big ones, can cause long-lived memory bloat in a Ruby process.

Here's a quick list of scenarios that could be causing a temporarily large allocation in your application:

- Opening large files, whether user attachments or something from the filesystem
- Large ActiveRecord queries - depending on the size of the ActiveRecord object, an array of just a few thousand could result in 100+ MB allocations.
- Especially large webserver responses or requests.
- Extremely complicated views - some views can result in tens of thousands of strings being allocated.

An easy way to confirm a hypothesis here is to use a tool like `gc_tracer`, covered in the memory profiling lesson, or write your own tool for logging `GC.stat`. Large amounts of free heap slots means that something is allocating huge amounts of memory which never gets released.

If this behavior of Ruby - not immediately releasing free heap slots - seems silly to you, it really isn't. Garbage collectors, to work at maximum efficiency, need predictable applications with somewhat predictable workload. Garbage collection will always be "expensive" - slowing your process by a few milliseconds (sometimes hundreds!) during collection. Ruby's algorithm for requesting more allocation from the operating system works just fine if your app has a stable, predictable memory load.

If you're suffering from one of the scenarios above and can't refactor it out of your application - reading large files or responses, especially - consider trying a streaming approach to the problem. One of the major advantages of streaming is that the entire

object or file in question doesn't need to all be in memory at a single point in time, reducing overall memory requirements.

For example, instead of trying to read a file of 1 million lines at once like this:

```
File.read("myfile.txt").each_line do |line|
  do_some_work(line)
end
```

... we can read the file line by line, never keeping more than a single line in memory, like this:

```
file = File.open("myfile.txt")
until file.eof?
  do_some_work(file.gets)
end
```

[Read up on Ruby's IO library for more.](#)

A similar effect can be accomplished by doing work in "batches" rather than all at once, like ActiveRecord's "find_each" method. If you're struggling with long responses, check out ActionController::Streaming, covered in the lesson on streaming and SSEs.

Oink

`oink` is a Rubygem that aims to help you manage memory bloat by reporting memory usage on a per-request basis. Install `oink` and it will start logging some memory-related stats. Later, you can plug these logs into `oink`'s included executable, and `oink` can tell you some interesting statistics, like which requests and controller actions increased your process' heap by the most amounts. Conceptually, it's not all that different from Skylight's allocation tracer.

Oink is an excellent tool for tracking down which controller actions may be allocating large amounts of memory, like mentioned above.

Gemfile Auditing

Dependencies have costs - and in Ruby, they can sometimes have significant memory costs.

My favorite tool for evaluating the memory cost of my Gemfile is Richard Schneeman's [derailed_benchmarks](#) . Derailed includes a benchmark for evaluating the memory cost of each gem in a Gemfile. Essentially what it does is open up a new ruby process, check how much memory it's using, then `require` s a single gem from your Gemfile, checks the memory usage, and subtracts to get the difference. This is a *static* benchmark, meaning your app is never actually run. Since it doesn't actually load your application, that means this benchmark isn't perfect - other parts of the gem may be loaded later by other parts of your application, or your particular *usage* of a gem may be imposing high memory costs.

As an example, I audited Rubygems.org's Gemfile using Derailed's static memory benchmark.

First, I add `derailed` to the Gemfile:

```
gem 'derailed', group: :development
```

...and run `bundle install` . Then, I run the benchmark:

```
bundle exec derailed bundle:mem
```

The output is pretty intense - in general, it looks like this:

```
TOP: 75.8516 MiB

delayed_job: 18.5234 MiB (Also required by: delayed/railtie, delayed_job_active_re
cord)
  delayed/performable_mailer: 18.2383 MiB
    mail: 18.207 MiB (Also required by: TOP)
      mime/types: 14.7344 MiB
```

The `TOP` bit is pretty self explanatory - the total memory usage of all of your gems added together.

We can tell that `delayed_job` takes up 18MB of memory at require time. Derailed also helpfully points out if any other gems require the same file. Indentation shows what other files are required - `delayed_job` requires `delayed/performable_mailer` , which requires 'mail', which requires `mime_types` .

The interesting thing in this particular output is `mime/types` taking up 14.7 MiB (that's 14.7 Mebibytes, or 15.4 Megabytes) of memory. That's a lot! This older version of `mime/types` used an inefficient method for storing its list of MIME types. This was fixed in `mime-types` version 2.6 - any version later than that will work.

All I had to do was add this to the top of the Gemfile:

```
# https://github.com/mime-types/ruby-mime-types/issues/94
# This can be removed once all gems depend on > 3.0
gem 'mime-types', '>= 2.6', require: 'mime/types/columnar'
```

Here's the full pull-request. This one small change saved over 30 MB of process memory!

Sometimes you also can find gems that use a lot of memory that aren't even used - I found that Rubygems.org had gems for Coffeescript and SASS in the Gemfile but never actually had any `.scss` or `.coffee` files in the project! [Removing those gems saved another 8MB of memory at startup.](#)

When evaluating the output of `bundle exec derailed bundle:mem`, ask yourself for each gem - do we really need this? Is there an alternative out there that has the same functionality at a lower memory cost?

For example, a popular gem for file uploads is `carrierwave`. `carrierwave` depends on `fog`, which installs several *dozen* other gems (at time of writing - they say they're going to fix this). However, an alternative, called `carrierwave-aws`, forgoes the `fog` dependency and just uses the `aws` gem directly. This saves almost 10 MB of memory at require time - nice!

If you can't find a lightweight gem to do a simple job, you may be stuck writing your own. However, the decision of whether or not to write your own code rather than use your own gem is beyond the scope of this guide. [RubyConf 2015 featured a good talk on the subject.](#)

jemalloc

I'm always on the hunt for "free wins" in performance - little changes that don't require me to rewrite any code or make big alterations, but still deliver big performance wins. There are few of these in the Ruby world - but I'm starting to come to around to one: jemalloc.

Ruby's calls to its memory allocator are abstracted - by default, they use glibc's `malloc`, but Ruby can use any `malloc(3)` compatible memory allocator. There are a lot out there, giving you quite a few choices, like Hoard, `tcmalloc` and `jemalloc`.

However, as far as I know, only one of these alternative allocators has received extensive review by the Ruby community - it's also arguably the strongest option available: `jemalloc`. [Developed by Facebook](#), `jemalloc` is a `malloc` implementation intended for modern multithreaded applications.

[Sam Saffron of Discourse](#) has confirmed that `jemalloc` can deliver 6-10% smaller memory heaps off the bat, with additional gains as heap sizes grow and applications continue running. One of the things `jemalloc` gets right is its management of memory fragmentation.

As of Ruby 2.3, you can compile Ruby with `jemalloc` with the `"--with-jemalloc"` option. For example, when using the `rbenv` ruby version manager, you can:

```
CONFIGURE_OPTS="--with-jemalloc" rbenv install 2.3.0
```

However, any previously compiled Ruby can be used with `jemalloc` by using the special environment variable `LD_PRELOAD`:

```
LD_PRELOAD=/usr/local/jemalloc/3.6.0/lib/libjemalloc.so ruby myscript.rb
```

I have taken over the maintenance of an experimental buildpack for Heroku that allows you to use `jemalloc` on Heroku applications - [you can try it out here](#).

For more information about memory allocators, there's an entire lesson on the subject in the Environment section of the Guide.

GC Parameters

Almost everyone reading this lesson will have experimented with GC parameters at some point in their Ruby career - most of us just tried to copy and paste some magical numbers from the internet and hoped they made things better. Usually, they don't. Ruby's garbage collector has been under active development from Ruby 2.1-2.3, and settings that worked on Ruby 2.1 may be complete anathema for the Ruby 2.3 algorithm. Overall, I would recommend *not* messing with these parameters. It simply isn't worth it for most applications anymore.

It's also worth noting that, when tuning garbage collection parameters, you're almost always working with a memory/CPU time tradeoff. More frequent garbage collection means less memory usage, but more time spent in garbage collection and a slower application (and vice versa).

For these reasons, I'm not going to provide garbage collection parameters here. I think the default settings are Good Enough for most applications - if you're really interested, check out [this awesome RubyConf talk](#) on how Ruby's GC works and then take a look at the variables defined in `gc.c`.

Checklist for Your App

- **Use Oink or `ps` to look for large allocations in your app.** Ruby is greedy - when it uses memory, it doesn't usually give it back to the operating system if it needs less memory later. This means short spikes turn into permanent bloat.
- **Audit your gemfile using `derailed_benchmarks`, looking for anything that require more than ~10MB of memory** Look to replace these bloated gems with lighter alternatives.
- **Reset any GC parameters you may have tweaked when upgrading Ruby versions.** The garbage collector has changed significantly from Ruby 2.0 to 2.3. I recommend not using them at all, but if you must - unset them each time you upgrade before reapplying them to make sure they're actually improving the situation.

Memory Leaks

Memory leaks. They're the thing that goes bump in the night. I'm going to acknowledge from the outset that this is probably one of the most important and most difficult topics in this course.

As an example, [there's a memory leak thread on the Puma GitHub issue tracker that's been open for over 3 years](#). The discussion goes back and forth - there's a leak, there isn't a leak, and no one can create an application that replicates the leaky behavior. It's a nightmare. Some of the smartest minds in the Ruby community are in that thread - including Koichi Sasada, Ruby core member and implementer of the CRuby virtual machine - and none can pin down exactly what's happening.

Part of the problem with memory leaks is that they can come from many sources:

- **Managed Ruby object leaks** - there are many ways to write Ruby code that leaks objects (or at least keeps objects around for longer than you may expect).
- **C-extension leaks** - frequently one of the hardest leak types to debug, C-extensions in your gems can leak. For example, leaks have been found in Markdown-producing gems that use C-extensions to convert text to Markdown. A leak was discovered in EventMachine's C interface with OpenSSL. C-extensions are particularly prone to leaking, of course, because C is not a memory-managed language.
- **Leaks in Ruby itself (the VM)** - Although a completely unsupported hunch, this is what I think has been happening to the Puma app server over the years. Threading in the Ruby VM was, until recently, a relatively untouched area of the language. Puma was the first popular threaded application server, and when it became widely adopted, bugs in the underlying thread implementation of Ruby were (or are being) exposed.

The real tough part is that each of these three sources of memory leaks requires a completely different approach and set of tools for diagnosing and fixing them. The last two sources (C-extensions and VM leaks) may simply be too complicated to diagnose and track down for many Ruby programmers.

But, before we get into how to fix leaks, we have to define what they are.

Ruling Out Bloat

I'd reckon that at least half of the time when a Ruby programmer *thinks* they have a memory leak, what they're really battling is memory *bloat* instead. First, go and read the lesson on memory bloat if you haven't already. There are many confounding factors which can make leaks and bloat look similar - they both involve memory growth, and in highly memory constrained environments (like 512MB Heroku dynos), they can both crash servers with out-of-memory errors.

What Leaks Look Like

A memory **leak** occurs when memory is **continually allocated and never released** by a process, even when that what is stored in that memory is no longer required. In Ruby web applications, **leaks are small and slow**, leaking just 40-300 bytes at a time. **Leaks never stop growing** and will continue to leak for as long the process runs.

Memory **bloat** occurs when a process **requires large amounts of memory**, and that memory is never released because it is in **continual use**. In Ruby web applications, **bloat usually occurs quickly**, represented as huge one-time spikes of 1 to even 100 megabytes. **Bloat eventually levels off**, with memory usage stabilizing.

Let's sum that up with a table:

| | | | | |
|---|-------|-----|-------|---------------------|
| Memory bloat Memory leaks | ————— | ——— | ————— | Allocated memory is |
| actually required Yes No Growth Fast and large Slow and steady Levels off | | | | |
| Eventually Never | | | | |

One of the most obvious differences between a leak and bloat is the time scale. All Ruby applications will grow in memory usage once they are initialized. This is simply the nature of Ruby and how we've architected our Rails frameworks. Not all of the files in an application are immediately required upon startup, and as these files are loaded in, memory usage increases. Caches are being populated as well - one that causes a lot of memory growth in Rails is the ActiveRecord statement cache. As SQL queries are executed, ActiveRecord caches the generated SQL statements so it can re-use them in the future. This is, of course, stored in memory - leading to slow growth over time.

In a typical production application, **memory usage should level off within 2-3 hours of the last restart**. Note that low-volume apps will take longer to level off. At the maximum, memory usage should level off within 24 hours in most applications. If not, consider that "one point towards the leak hypothesis."

Back Off The Memory Pressure To See Clearly

One reason Ruby developers confuse bloats for leaks is that they never see the "tail end" of their memory usage graph because the process crashes quickly or is restarted automatically by a "worker killer" or other process monitor.

If you're unsure if you've got a leak or bloat, **try allowing your Ruby processes to run for at least 24 hours without a restart**. To do this, you need to turn off any process monitors that will automatically restart processes, and you need to reduce memory pressure on your application.

If you're running multiple workers or Unicorn processes per server, dial that down to just 1. Scale up the total number of servers if necessary to make up for the lost capacity. If necessary, you may also need to reduce the amount of threads you're using with a threaded webserver.

Alternatively, you could increase the memory limits of your server temporarily - if on Heroku, upgrade to a Performance dyno for 24 hours (without changing your worker count). If on AWS, change your instance type.

Either way, try to increase the amount of memory headroom any single process has to deal with. Keep increasing the amount of headroom until the process no longer runs out of memory and levels off after 24 hours.

If, even after reducing yourself to just one Ruby process per server, you're still seeing memory growth after 24 hours, it's time to move on to the next stage of diagnosis.

Tools We Can Use To Diagnose Leaks

At this point, you're going to want to break out the tools.

When diagnosing leaks, it's easy to go down the rabbit hole of tooling. One minute, you're just twiddling number-of-processes-per-dyno, and the next you're knee deep in Valgrind trying to guess which memory addresses correspond to which esoteric library you're using. No really - many memory leaks are solved by extremely esoteric means, especially those buried deep in C-extensions.

Instead, rather than *exactly* pinpointing a memory leak's source by introspecting the memory itself, it's far easier to **have a long-running production memory metric in production**. If you can graph your production memory usage for months at a time, it's far easier to look back and identify which deploys introduced a problem. If you upgraded a gem a week ago, and ever since your memory has been steadily growing until a restart, you can probably guess it's related to that gem. **An ounce of prevention is**

worth a pound of cure when it comes to memory leaks in this sense. If you're not tracking memory usage in production, and can dig back into logs ~2-3 months old, you're inviting a leak scenario that will require the muckiest of tools to debug.

New Relic provides this sort of long-lived memory introspection by default - it's available in their "Ruby VMs" tab. Heroku's 24 hour memory tracking isn't enough - you need to be able to step back at least a week at a time, unless you plan on checking your app for memory leaks every single day.

Reproduce Locally

Our next step, now that we're pretty sure that we've got a leak, is to try to reproduce it locally. At this point, we're still not exactly sure what's causing the leak, only that we definitely have a leak.

Use Siege To Simulate Load

We'll set up `siege`, the load testing utility, to pound a list of URLs that we think may be causing a leak. Just write out a text file with a list of the URLs you want to hit - it might look something like this:

```
http://localhost:3000/  
http://localhost:3000/gems  
http://localhost:3000/gems/1  
http://localhost:3000/gems/2
```

Then, we can start siege with:

```
siege -c 32 -f urls.txt -t 5M
```

`siege` will pound your local server instance over and over with this list of URLs. In this case, we're using 32 threaded workers and the test will last for 5 minutes. The test probably needs to be run for longer than you think - make sure you're getting at least 10,000 requests completed during the test timeframe (it displays in the output as "transactions completed").

The design of this test is important - rather than manually triggering requests to the webapp, we're doing as many as possible so that we'll trigger lots of garbage collections. While it's doing this, we're going to watch for a slow leak.

Count objects with GC.stat and ObjectSpace

While the `siege` test is running, we're going to want to log some critical numbers.

- **RSS memory usage.** This is what we're trying to decrease. If this number levels off after 10k requests or more, we probably don't have a leak.
- `GC.stat[:heap_live_slots]` This is the number of Ruby's memory slots that are occupied by objects. If this number stays flat while RSS is increasing, we *probably* have a C-extension leak, though we may be just modifying a single object and making it larger over time (unlikely).
- `GC.stat[:heap_free_slots]` If this number is large (~10% of `heap_live_slots`), it's an indicator of bloat. Ruby does not release heap memory back to the operating system - high numbers of heap free slots is an indicator that large amounts of memory are being required to accomplish a task, and then not used again. See the memory bloat lesson for more.
- `GC.count_objects` This hash, covered in the memory profiling lesson, will show the number of each type of object currently live in your Ruby process. If one of these grows unbounded, we probably have a leak in the Ruby memory space.

We can track these numbers with a simple memory-logging thread:


```

# config/initializers/memlog.rb
Thread.new do
  logger = Logger.new('mem_log.txt')
  logger.formatter = proc { |sev, date, prog, msg| msg }

  headers = [
    "RSS",
    "Live slots",
    "Free slots",
    ObjectSpace.count_objects.keys
  ].flatten

  logger.info headers.join(",")

  while true
    pid = Process.pid
    rss = `ps -eo pid,rss | grep #{pid} | awk '{print $2}'`
    memory_info = [
      rss.strip,
      GC.stat[:heap_live_slots],
      GC.stat[:heap_free_slots],
      ObjectSpace.count_objects.values
    ].flatten

    logger.info memory_info.join(",")
    logger.info "\n"

    sleep 5
  end
end

```

Add that to your config/initializers directory, and you'll get a comma-separated memory log in the root of your application. Run the test with `siege`, gather 10-50k requests, and then load this log into Excel and start doing some analysis.

Zeroing In on the Leak

It's at this point where things start to get *really* fuzzy. As mentioned at the beginning of the lesson, there are a lot of points where your application can leak and they'll all require different tooling.

Best Case Scenario: A Ruby Object Leak

If you notice *heap live slots* increasing, *heap free slots* remaining low (below a few thousand slots), and *RSS increasing*, you probably have a leak in the Ruby object space. This is good news, as it's probably the easiest to track down.

You have a Ruby object that is leaking somewhere - perhaps references are being retained to the object even though it's no longer needed.

Looking for leaks with `memory_profiler`

To get a better idea of where the leak is coming from, dig in with the `memory_profiler` gem.

If you're not sure *exactly* where the leak is occurring, use `memory_profiler` with `rack-mini-profiler`. When both of these gems are installed, you can see a report of retained objects on a per-request basis. Look for retained objects by location in the `memory_profiler` report for actions you suspect are leaking memory.

If you're reasonably certain that the leak is occurring with your use of a certain gem, write a script that uses the gem stand-alone and wrap its execution in `memory_profiler`. This is *exactly* what Sam Saffron, `memory_profiler` and `rack-mini-profiler` did to discover a memory leak in `therubyracer`:

```
ENV['RAILS_ENV'] = 'production'
require 'memory_profiler'

# this requires all the files in your Rails app. Normally lives in config.ru
require File.expand_path("../config/environment", __FILE__)

# Warmup. This is important, and similar to performance
# benchmarking. We want to run the code at *least* once before
# profiling it to make sure we're not profiling the "first run"
# behavior, which may be more complicated than following runs.
# PrettyText is a module Sam uses to convert Markdown to HTML. It
# uses TheRubyRacer internally.
3.times{PrettyText.cook("hello world")}

MemoryProfiler.report do
  50.times{PrettyText.cook("hello world")}
end.pretty_print
```

You could use this snippet as a general pattern when testing for leaks in your own application. Look at retained objects by location in the output to find areas where you're probably leaking memory.

For more about `memory_profiler`, see the memory profiling lesson.

Known Leaky Gems

It's a good idea to make sure you're not using any gems which are known to leak memory. [There is a community maintained project on Github that keeps a list of known leaky gems](#), though you should also search "memory leak" in the issues section of gems you suspect are causing you trouble.

Some notable leaky gems include:

- Celluloid, versions between 0.16.0 and 0.17.2
- grape < 0.2.5
- oj < 2.12.4
- redcarpet < 3.3.3
- sidekiq < 3.5.1
- therubyracer < 0.12.2

Not-So-Great: A C-Extension Leak

If *heap live slots* and *heap free slots* are remaining constant while *RSS is increasing*, you probably have a leak in the C-extensions of one of your gems.

This is a difficult situation to debug. If you don't have experience with C programming, you may find locating these leaks difficult. If you can, you may just want to install a worker killer (see below) and hope the problem doesn't get worse. Otherwise, get prepared for a multi-week battle against this leak.

For the strong of heart, there are two good tools I know of for tracking down C-extension leaks.

Heap Dumps

Heap dumps involve turning on memory allocation tracing, dumping the entire Ruby object heap to a file, and then comparing these dumps against each other to try to get a sense of what objects are being added to the heap (and retained).

Heap dumping is a complicated subject, and the APIs for doing this may change. If you suspect you have a C-extension leak and want to try your hand at heap dumping, I can point you to the following online resources:

- <http://blog.codeship.com/the-definitive-guide-to-ruby-heap-dumps-part-i/>
- <https://github.com/tenderlove/heap-analyzer>
- <https://samsaffron.com/archive/2015/03/31/debugging-memory-leaks-in-ruby>
- <http://gavinmiller.io/2015/debugging-memory-leaks-on-heroku/>

jemalloc Introspection

An interesting alternative approach is to use the instrumentation available in your memory allocator - `jemalloc` has great facilities for this. For more about installing and using `jemalloc` with Ruby, see the Alternative Allocators lesson near the end of the Course.

When Ruby is running with `jemalloc`, we can use the `MALLOC_CONF` environment variable to trigger some of `jemalloc`'s built-in profiling.

```
export MALLOC_CONF='prof_leak:true,lg_prof_sample:0,prof_final:true'
```

As an example, the above configuration will log `jemalloc`'s heap at exit - you can use this dump to get an idea of where memory may be leaking, because memory that's still around at exit means it was probably forgotten by whatever was supposed to free it up!

For more about using `jemalloc` to track down memory leaks, see this story about tracking down a C-extension leak by [Oleg Dashevskii](#) and [the jemalloc wiki entry](#).

Absolutely Awful: A Ruby VM Leak

If *heap live slots* and *heap free slots* are remaining constant while *RSS is increasing*, and you can't find a leak in your gems or C-extensions, you may be experiencing a leak in the Ruby VM itself.

At this point, I must point you to other resources, like the ruby-core mailing list. I am not a Ruby core developer, nor do I ever wish to be - and leaks at this scale are simply beyond my technical ability.

For a story about tracking down a memory leak in the Ruby VM, [check out Alexey Gaziev's tale of fixing a threading-related bug in Ruby 2.2](#).

Giving Up: Worker-Killers

If you can't fix the problem, you can put a band-aid on it. By simply restarting your application as soon as it starts to use too much memory, you can at least prevent the usage of swap memory and out-of-memory errors from crashing or slowing down your application. I recommend these gems only as a "last resort" or as a "can't fix now, will fix later":

- `puma_worker_killer`
- `'unicorn_worker_killer'`
- Phusion Passenger Enterprise can enforce a maximum per-process memory usage by itself.

Checklist for Your App

- **Get a long-running memory log.** If you're not logging memory usage over a week or month long timeframe, you're losing valuable data that could be used when tracking down memory leaks. Being able to track memory usage against deploy times is absolutely critical to avoid tons of hard, dirty debugging work.

Common ActiveRecord Pitfalls

ActiveRecord is a double-edged sword. Its great strength - its unparalleled ease-of-use - is also the thing that makes it likely to cut you. ActiveRecord makes it easy to grind your server to a halt: just a few lines of ActiveRecord queries can easily grind any Rails process to a halt.

Some would throw such a sharp tool out of the toolbox altogether - reaching instead for alternatives in the Ruby ORM world like [Sequel](#). But I still prefer ActiveRecord's expansive and powerful API. Of all the existing Ruby ORMs, I find its the one that lets me write the least amount of SQL to accomplish any given task - and I hate writing SQL.

This lesson will cover the most common performance-related issues when dealing with ActiveRecord. This is inevitably linked to the lesson on databases, but we'll be focusing on the Ruby layer here.

Operations on Many Records - `find_each` and `in_batches`

Here's a really common pitfall that happens all the time in background jobs, when many records are being processed at once. Perhaps you've got a batch job for updating a bunch of Subscription objects. On the first of the month, their number of "months spent as customer" needs to be incremented by one (silly example, I think, but work with me).

Here's one way that might look:

```
Subscription.all.each do |sub|
  sub.months_as_customer += 1
  sub.save!
end
```

There's a major problem with this approach, and indeed, *any* ActiveRecord query that uses `all` or any scope that could return many records. Before that `each` can enumerate through all the records, the query must be executed and *all of those records must be loaded into memory*. Ouch!

Here's an example from Rubygems.org:

```
gem = Rubygem.limit(1).order("RANDOM()").first
require 'objspace'
ObjectSpace.memsize_of(gem)
#=> 120
all_gems = Rubygem.all
ObjectSpace.memsize_of_all(Rubygem)
#=> 15313024
```

For more about `memsize_of`, check out the lesson on memory profiling.

Ouch - it takes at *least* 15 megabytes of memory to load up all 120,000 Rubygems in the database. This *doesn't count* all of these objects attributes either, which also have to be instantiated.

As described in the lesson on memory bloat, huge memory allocations like this can cause a spike in memory usage that the process never really releases again. A Ruby process might be using 100MB before you run this query, and then afterward it sits at 200MB, even though it's only using 100MB internally.

The alternative is to *not* load all of these records at once. Let's say we've decided that Rubygems aren't LOUD enough, and all gem names must be in ALL CAPS. The naive implementation would be:

```
Rubygem.all.each { | g |
  g.name.upcase!
  g.save
end
```

Running this code on my local machine immediately balloons process memory from 100MB to 330MB! Ouch!

Using `ActiveRecord#find_each`, though, we can shrink this number considerably. Instead of loading *all* of those records at once, `find_each` loads them in batches. By default, this batch size is 1000 records.

```
Rubygem.all.find_each do |g|
  g.name.upcase!
  g.save
end
```

This simple change keeps process memory at a low ~120MB, a 210MB improvement over our previous example! `find_each` accepts batch sizes and can even start at different points in the query (i.e., start at the 2000th record in this batch of 10000).

There's also a new method in Rails 5 called `in_batches`, which yields the `ActiveRecord::Relation` of the entire batch to the block, rather than a single record:

```
Rubygem.where("downloads > 100").in_batches do |relation|
  relation.update_all(popular_gem: true)
end
```

Good God Man, Stop with the N+1s!

N+1 queries. They're sort of a pet peeve of mine - I feel like they're harped on in nearly every "speed up your Rails app!" blog post, so I almost don't want to hype them up again here. You should know better by now, right?

Right off the bat, I'll discuss a bit of advice given for this problem that I *don't* agree with: I don't use `bullet`.

Bullet is a gem that is supposed to identify N+1 queries and opportunities to use eager loading, which I'll get to in a second. This sounds great - just do what the gem tells you to do, right?

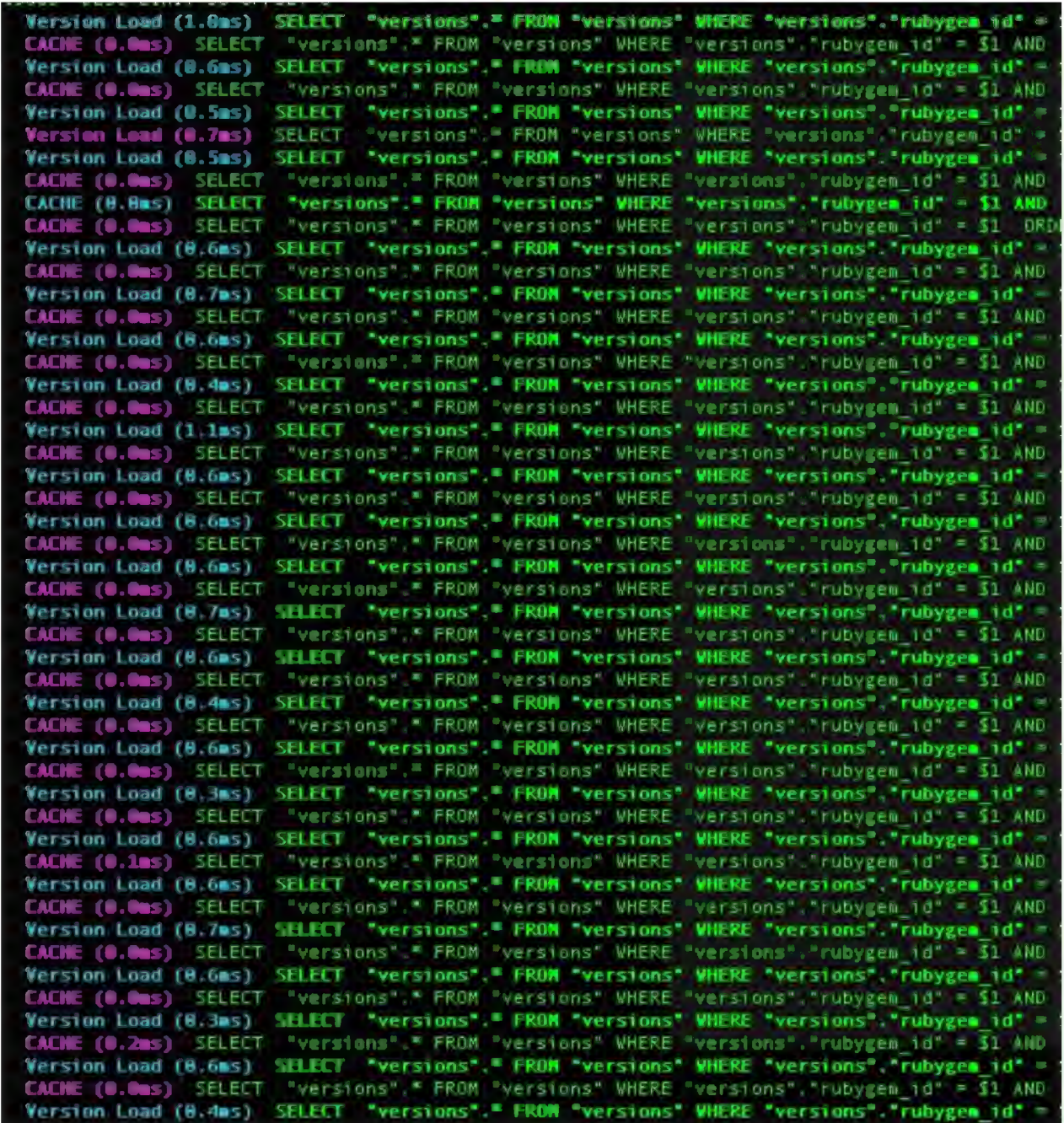
Using `bullet` ends up with a couple of problems in practice:

- `bullet` can't work with complicated stacktraces, meaning it sometimes misses N+1 queries caused by third party gems or Rails engines. Or, it may identify that an N+1 query is happening but it can't tell you where.
- Not every "N+1"-like query is avoidable. I don't like using code linters, like `bullet` or `rubocop`, unless I can 100% comply with its guidelines. Once you start ignoring a linters warnings, it quickly turns into a case of "the boy who cried wolf" - you start ignoring all of them.
- `bullet` is eager (hah!) to suggest eager loading. In many cases, this is not appropriate. I'll get to this bit in a second.
- `bullet` discourages the use of production-like data in development - rather than learning to identify N+1 problems by watching development logs, you just wait for the (imperfect) tool to tell you what to fix and optimize.

I'd rather we just all learned how to identify N+1 issues ourselves - maybe I'm a bit of an old grump on this point, but you're free to try `bullet` and see for yourself.

How do you find N+1 queries "the hard way"? By reading your logs, of course!

The nice thing is that they're extremely easy to identify when you have production-like data loaded in to your local, development database. As an example, I loaded up the Rubygems.org production data dump into my local database, and looked at the Rubygem index. Here's what I saw:



```

Version Load (1.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.6as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.5as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
Version Load (0.7as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
Version Load (0.5as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
CACHE (0.8as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.6as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.7as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.6as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.4as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (1.1as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.6as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.6as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.6as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
WHERE "versions"."rubygem_id" = $1 AND
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.7as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.6as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.4as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.6as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
WHERE "versions"."rubygem_id" = $1 AND
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.3as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.6as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.7as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.6as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.3as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.2as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.6as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =
CACHE (0.0as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND
Version Load (0.4as) SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" =

```

This is what an N+1 looks like - many similar SQL queries, repeating. They're extremely common on `index` actions, but can be present anywhere a model collection is iterated over.

This method of watching the logs means you need to have production-like quantities of data in the database. Often, developers use empty or nearly empty databases in development, which means they'll never see `MyModel.all` return 10000 records instead of 5 or 10, missing the N+1 query in the process.

Now that I know I have an N+1 query, I have to figure out where the SQL query is coming from. To do this, I have a small piece of code I drop into `config/initializers` that logs where SQL queries are generated. It's not 100% accurate all the time, but it gives me a good starting point.

```
# config/initializers/query_source.rb
module LogQuerySource
  def debug(*args, &block)
    return unless super

    backtrace = Rails.backtrace_cleaner.clean caller

    relevant_caller_line = backtrace.detect do |caller_line|
      !caller_line.include?('/initializers/')
    end

    if relevant_caller_line
      logger.debug("  ↳ #{relevant_caller_line.sub("#{ Rails.root }/", '')}")
    end
  end
end
ActiveRecord::LogSubscriber.send :prepend, LogQuerySource if Rails.env.development?
```

After adding that initializer and restarting my server, I see the following under each of my suspected N+1 queries:

```
↳ app/models/version.rb:112:in `most_recent'
```

You can also use `rack-mini-profiler` for this - see the lesson for more information.

So something is calling `some_rubygem_version.most_recent` on every gem on this index page. After some digging, I find the offending line in a partial:

```
<p class="gems__gem__desc t-text"><%= short_info(rubygem.versions.most_recent) %></p>
```


This partial is rendered for every gem in the `#index`. To test my hypothesis, I simply remove this line and see if the N+1 goes away. Success!

The Rubygem model is loaded in this view with this:

```
@gems = Rubygem.letter(@letter).by_downloads.paginate(page: @page)
```

...and the `most_recent` method looks like this:

```
def self.most_recent
  latest.find_by(platform: 'ruby') || latest.order(number: :desc).first || last
end
```

The problem here is the `find_by` call in `most_recent` - this will *always* trigger a query, even if we add some eager loading to the controller!

Instead of doing using ActiveRecord methods that trigger SQL queries, we're going to *rewrite* this method to use regular Arrays and Enumerable methods. I ended up adding a method the Rubygem model that looked like this:

```
def most_recent_version
  latest = versions.select(&:latest).sort_by(&:number)
  latest_for_cruby = latest.select { |v| v.platform == "ruby" }

  if latest_for_cruby.any?
    latest_for_cruby.last
  elsif latest.any?
    latest.last
  else
    versions.last
  end
end
```

Note that this version is a lot longer than the previous `Version.most_recent` method. Notice also that I've basically just replaced ActiveRecord query methods with Enumerable equivalents. [You can see the final pull request and discussion here.](#)

Queries in Controllers and Scopes Only!

The particular example above from Rubygems.org is an instance of an extremely common pattern in Rails applications. Methods on a model trigger SQL queries (by using the ActiveRecord API), and then those methods get called in the view. Inevitably, they end up being used in a partial or something that gets iterated for every element in a collection, and bam - N+1.

I've seen this so often that I'm willing to generalize a rule:

Do not use ActiveRecord query methods inside models. Use them only in controllers and helpers. ActiveRecord scopes are excepted from this rule.

In a way, this makes intuitive sense: a model is intended to represent a *single instance* of the database row in the ActiveRecord pattern. The responsibility for querying and organizing the data generally comes from elsewhere - the controller.

Here's a full list of ActiveRecord methods that could generate a SQL query, and therefore an N+1 if used on an element of a larger collection:

- bind
- create_with
- distinct
- eager_load
- extending
- from
- group
- having
- includes
- joins
- limit
- lock
- none
- offset
- order
- preload
- readonly
- references
- reorder
- reverse_order
- select
- uniq

- where

Replace Query Methods With Enumerable

In the Rubygems example, we also replaced ActiveRecord query methods with their Enumerable equivalents. This basically moved the work of selecting the latest Rubygem version from the database into our Ruby process.

That probably doesn't make sense - normally, we would want the database to do *more* work, not less, right? The database is surely faster than whatever we can do in Ruby? Sometimes that's true - but when there's N+1's happening, usually it isn't.

When encountering an N+1, the solution usually lies in replacing ActiveRecord query methods (listed above) with Enumerable equivalents - combinations of `select`, `reject`, and `sort_by`.

The reason why this is faster isn't because Ruby is faster than your DB - its because we're instantiating fewer ActiveRecord objects. To return to the Rubygems example, consider:

```
def self.most_recent
  latest.find_by(platform: 'ruby') || latest.order(number: :desc).first || last
end
```

In the worst case, this triggers 3 SQL queries:

```
Version Load (0.6ms)  SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND "versions"."latest" = $2 AND "versions"."platform" = $3 LIMIT 1  [["rubygem_id", 31121], ["latest", "t"], ["platform", "ruby"]]
Version Load (1.9ms)  SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 AND "versions"."latest" = $2 ORDER BY "versions"."number" DESC LIMIT 1  [["rubygem_id", 31121], ["latest", "t"]]
Version Load (1.2ms)  SELECT "versions".* FROM "versions" WHERE "versions"."rubygem_id" = $1 ORDER BY "versions"."id" DESC LIMIT 1  [["rubygem_id", 31121]]
```

For each one of these queries, we have to instantiate *new* ActiveRecord objects for each of the returned rows. This takes a lot of time.

Our optimized version only uses 1 set of ActiveRecord objects, and no new ones are instantiated when calling our optimized method.

There's a caveat to this approach - there are going to be times when doing the work in the database and then instantiating the ActiveRecord object will be faster than searching the larger collection with Ruby. Only testing and benchmarking will tell you when this tradeoff is occurring - be sure to benchmark any changes.

Select Only What You Need

Another way to cut down on memory usage (and time!) is to select only portions of the entire model - just the ones that you need.

What do I mean? Say you've got a model called `car`, and this model has several attributes, but one of them is `service_record`. The `service_record` is an import from a legacy system - it's just a big text dump of the car's maintenance history, probably in some obscure weird format. We only use this attribute in a limited part of the application.

However, each `car`'s `service_record` will be instantiated whenever we load up a `Car` object. If, on average, `service_records` are fairly large (say, a few KB), this could impose a massive memory tax on any actions that work with many `car` objects. And we don't even *use* the `service_record` attribute often!

The solution lies in an often-overlooked part of the typical ActiveRecord query. Take a look:

```
Car Load (193.7ms) SELECT "cars".* FROM "cars"
```

The interesting part here is the asterisk - we're selecting *all* columns from the `Car` table. If we don't select certain columns, ActiveRecord won't instantiate the corresponding attribute. This saves memory in Ruby, time in the database, and time when instantiating the ActiveRecord object!

```
Rubygem.all.select(:name, :id)
# Rubygem Load (80.4ms) SELECT "rubygems"."name", "rubygems"."id" FROM "rubygems"
```

I would only use `select` as a performance optimization when I knew I had a slow SQL query that returned many rows. Don't reach for this one too early, as it increases coupling between your controllers and views.

If you try to access attributes that you haven't `select` ed, ActiveRecord will raise an `ActiveModel::MissingAttributeError` :

```
Car.select(:make, :model).first.color # Boom!
```

Note that `select` returns ActiveRecord objects - they just don't have all of their attributes loaded. For example:

```
irb(main):001:0> Rubygem.all.select(:name).to_a.first
Rubygem Load (58.0ms)  SELECT "rubygems"."name" FROM "rubygems"
=> #<Rubygem id: nil, name: "zyps">
```

So how do you know what columns to select? Rails 4.2 added the awesome `accessed_fields` method, which allows us to show what columns we actually *used* in any given view. Here's the example straight from the Ruby docs:

```
class PostsController < ActionController::Base
  after_action :print_accessed_fields, only: :index

  def index
    @posts = Post.all
  end

  private

  def print_accessed_fields
    p @posts.first.accessed_fields
  end
end
```

You could then update your `Post.all` query to `select` only the actual fields used in the view.

If we don't actually want ActiveRecord objects, and instead just want an array of values, it's far faster to use `pluck` :

```
irb(main):001:0> Rubygem.all.pluck(:name).first
Rubygem Load (58.0ms)  SELECT "rubygems"."name" FROM "rubygems"
=> "zyps"
```

This makes sense - instead of initializing thousands of complicated ActiveRecord objects, we just initialize a few simple, primitive objects.

Take it Easy With Lazy Loading

It's worth familiarizing yourself with ActiveRecord's three different eager loading methods. Each one will proactively fetch associations from the database, rather than causing another query when you try to access them later:

- `eager_load` will always use `LEFT OUTER JOIN` when eager loading the model associations.
- `preload` generates an extra query for each model specified in its arguments. These queries are then combined in Ruby, making it the slowest of all the eager loading methods.
- `includes` is *supposed* to "decide for you" if it's appropriate to use `eager_load` or `preload` for loading this particular set of ActiveRecord associations. Ideally, you would use `eager_load` everywhere possible, because it generates a single SQL query, not one for each association.

Generally, you can just use `includes` everywhere, but if you're not seeing the results you want, try forcing a join with `eager_load`.

Eager loading is great, but sometimes, too much can be a bad thing.

If you only paid attention to `bullet` or just listened to most advice for avoiding N+1s, you might think you should be dropping `includes` and other eager loading methods into every controller method you can get your hands on. That isn't the case.

Again, it comes down to ActiveRecord object instantiation. How many new ActiveRecord objects does this query create? With complicated calls to `includes`, this can often balloon.

For example, consider this:

```
Car.all.includes(:drivers, { parts: :vendors }, :log_messages)
```

How many ActiveRecord objects might get instantiated here?

The answer is:

```
# Cars * ( avg # drivers/car + avg log messages/car + average parts/car * ( average parts/vendor) )
```

Each eager load increases the number of instantiated objects, and in turn slows down the query. If these objects aren't used, you're potentially slowing down the query unnecessarily. Note how nested eager loads (parts and vendors in the example above) can *really* increase the number of objects instantiated.

Be careful with nesting in your eager loads, and always test with production-like data to see if `includes` is really speeding up your overall performance.

Do Math In The Database

I advocated removing ActiveRecord query methods from the model above. I even advocated for, in some cases, trying to do more work with Enumerable and small collections of Ruby objects rather than going to the database to perform the same work.

However, when operating on large collections of objects (1000s or more), it is almost always faster to try to do operations in the database. Nowhere is this more apparent than when doing mathematical calculations, like averages:

```
Rubygem.average(:downloads).to_i  
# (28.3ms)  SELECT AVG("rubygems"."downloads") FROM "rubygems"  
# => 8722
```

Grabbing each individual Rubygem record and calculating the average would take *ages* - by doing it all in the database, we're saving tons of time and memory.

In addition to `average`, there are several of these methods in

`ActiveRecord::Calculations`:

- `average`
- `calculate`
- `count`
- `ids`
- `maximum`
- `minimum`
- `pluck`
- `sum`

Note that this still fits with my original guideline: avoiding instantiating ActiveRecord objects. By using `ActiveRecord::Calculations`, we can usually instantiate *none*!

Don't Use Many Queries When One Will Do

Another area where applications typically create too many ActiveRecord objects is when doing mass updates. We already talked about using `find_each`, which solves the memory problem associated with these updates, but it's still slow to go through 10,000 rows individually, one-by-one.

In this case, you can drop down to the database.

When creating many records, the clear winner here seems to be the `activerecord-import` gem. I'll just quote directly from their README to give you an idea of the impact:

Say you had a schema like this:

- Publishers have Books
- Books have Reviews

and you wanted to bulk insert 100 new publishers with 10K books and 3 reviews per book. This library will follow the associations down and generate only 3 SQL insert statements - one for the publishers, one for the books, and one for the reviews.

In contrast, the standard ActiveRecord save would generate 100 insert statements for the publishers, then it would visit each publisher and save all the books: $100 \times 10,000 = 1,000,000$ SQL insert statements and then the reviews: $100 \times 10,000 \times 3 = 3M$ SQL insert statements,

That would be about 4M SQL insert statements vs 3, which results in vastly improved performance. In our case, it converted an 18 hour batch process to < 2 hours.

In addition, for other operations, look to methods on `ActiveRecord::Relation` to operate on many records at once:

- `update_all` performs a single SQL UPDATE to change the attributes of many rows at once.
- `destroy_all` can destroy many rows at once, rather than generating a single "DELETE" query for many rows.

Checklist for Your App

- **Any instances of `SomeActiveRecordModel.all.each` should be replaced with `SomeActiveRecordModel.find_each` OR `SomeActiveRecordModel.in_batches`.** This batches the records instead of loading them all at once - reducing memory bloat and

heap size.

- **Use production-like data in development.** Using production-size data in development makes N+1 problems much more obvious. Either set up a process for sanitizing production data or set up a `seeds.rb` that creates production-like quantities in the database.
- **Pay attention to your development logs to look for N+1 queries.** I prefer using the included query-logging middleware. `rack-mini-profiler` also works well for this purpose.
- **ActiveRecord instance methods should not use query methods - where, find, etc.** This inevitably causes N+1 problems when these methods are used later in a view. Use query methods in scopes (class methods) and controllers only.
- **When a query is particularly slow, use select to only load the columns you need.** If a particularly large database query is slowing a page load down, use `select` to use only the columns you need for the view. This will decrease the number of objects allocated, speeding up the view and decreasing its memory impact.
- **Don't eager load more than a few models at a time.** Eager loading for ActiveRecord queries is great, but increases the number of objects instantiated. If you're eager loading more than a few models, consider simplifying the view.
- **Do mathematical calculations in the database.** Sums, averages and more can be calculated in the database. Don't iterate through ActiveRecord models to calculate data.
- **Insertion, deletion and updating should be done in a single query where possible.** You don't need 10,000 queries to update 10,000 records. Investigate the `activerecord-import` gem.

Lab: ActiveRecord

This lab requires some extra files. To follow along, download the source code for the course and navigate to this lesson. The source code is available on GitHub (you received an invitation) or on Gumroad (in the ZIP archive).

Exercise 1

Included is a simple and slow ActiveRecord script - `lab/script.rb`. Speed up its execution.

You may want to use `time` as a benchmark: `time ruby lab/script.rb` .

Backgrounding Work - Why Do Now What You Can Do Later?

Your user has just finished signing up for YourSweetService. They click the “submit” button on the signup form, and wait. And wait. And wait.

What are they waiting for?

Perhaps they’re waiting for a signup email to be sent to them. Well, if you think about it - that’s sort of a silly thing to wait for, isn’t it? The email will get to them *eventually*. Why are they waiting?

They’re waiting because your code probably looks something like this:

```
class UsersController < ApplicationController
  def create
    user = User.new(params[:user])
    if user.save
      redirect_to signup_success_url
    else
      render :new
    end
  end
end

class User
  after_commit :send_signup_email

  def send_signup_email
    UserMailer.signup_email(self).deliver
  end
end
```

There’s a problem here - redirecting the user won’t occur until UserMailer has finished delivering the signup email. That may take a while - rendering an email takes time, and UserMailer will still have to make a network connection to your email provider to send the email. All of this might add up to a whole second or so.

Controller actions like this one - that *always* take more than ~300 milliseconds to execute, and *may* take much longer, depending on network conditions or an external service, are a performance anti pattern for a number of reasons:

- **They're unpredictable.** This is especially true if you're pinging external service providers over HTTP - say, Stripe for payment processing or Mailchimp for email. You never know when it will take 5 seconds instead of 300 milliseconds to complete a job. The action's speed varies wildly depending on the time of day or the current state of internet traffic.
- They're usually **not designed for failure**. Consider the above example - what happens if UserMailer fails?
- Other **requests will "back up"** behind this one. Requests with unusually high response times will cause other requests to "back up" in the queue of waiting-to-be-processed responses, increasing the response time of *those* otherwise fast requests.

When should a web transaction be moved to the background?

- **The action *always* takes more than your average response time to complete.** Some work just takes a long time - for example, transcoding video files or generating PDFs. That sort of work should always be done in the background. Use your average response time as a rule of thumb - if it *always* takes longer than ~150% of your average response time and you can't make it any faster, background it.
- **The action contacts an external service over the network.** Networks are not reliable. A request may take 100 milliseconds, it may take 100 seconds. Not to mention that *services* are unreliable - Mailchimp may take 100 milliseconds to process your email, it may never process it at all. It's far better to design for latency and failure rather than just hope it doesn't happen - background jobs let us do this.
- **The user does not care if the work is completed immediately.** Users don't need to wait for an email to send to see that their signup completed. If their credit card has already been authorized for the amount you wish to charge, they don't need to wait around for that charge to go through. If you're doing work during a response that they user doesn't need done right away, you're wasting their time - do it later!

Moving work out of the request/response cycle almost by definition will decrease your average response times, and also contribute to making them less variable and more predictable. This is awesome and helps make our apps more scalable, all while improving end-user experience.

Patterns

Here are some assorted patterns for safe, performant and reliable background job processing:

Idempotency - what happens if I retry this?

In computer science, the term idempotent is used more comprehensively to describe an operation that will produce the same results if executed once or multiple times.

For any given background job, you should be able to run it twice (or really an infinite number of times) and still get the result you desire. For example, here's a typical non-idempotent background job (implemented in `ActiveJob`):

```
class UserSignupMailJob < ActiveJob::Base
  queue_as :default

  def perform(user)
    UserMailer.signup_email(to: user).deliver
  end
end
```

If this job runs twice, we'll send two emails to the user. This is not a good thing - our user doesn't need to get the same "Thanks for signing up!" email twice!

Here's where it gets interesting - when writing background jobs, **we always must assume that it's possible an enqueued job may be executed twice**. Background job processors cannot *fully guarantee* that a job will not be executed twice, and even when they say they do, there's usually ways (like unplugging servers from the wall) that they still can.

The solution is usually to add some kind of mechanism that checks to see if the work has already been done. The most reliable way to do this is with a row-level database lock:

```

class UserSignupMailJob < ActiveJob::Base
  queue_as :default

  around_perform do |job, block|
    user = job.arguments.first
    user.with_lock do
      return if user.signup_email_sent
      if block.call
        user.update_attributes(signup_email_sent: true)
      else
        retry_now # or implement your own backoff procedure
      end
    end
  end

  def perform(user)
    UserMailer.signup_email(to: user).deliver
  end
end

```

That `around_perform` block protects against a number of scenarios:

- If for some reason the job is enqueued with the same user twice, the email will only be sent once. The first job will modify the user's `signup_email_sent` attribute, and the second job will exit after `return if user.signup_email_sent`.
- If two jobs with the same user are executed at the same time (possible if the user rapidly hit refresh and sent a form twice, for example), the `with_lock` block will stop two workers from operating on the same user at the same time. The lock will block the second worker until the first one has finished.
- If, for some reason, our `deliver` method fails (delivery failure, for example), we set up the job to be retried.

This seems like a lot of work, but you won't need to do this for every job. Some work is naturally idempotent - for example, if we wanted to change a `Car` object's `color` attribute to "red", we could enqueue that job as many times as we wanted and the end result would still be the same. Just ask yourself - what happens if this job is run twice with the same arguments? Note that this is pretty easy to write a test for too!

Scale your workers according to queue depth and Little's Law

Recall our lesson on scaling - scaling background job workers is no different than scaling web workers. If your queue depth is zero, adding additional workers is a waste of resources. Any “auto-scaling” solution you implement should base its decisions based on queue depths - not job execution time.

In addition, consider that the entire nature of background jobs is that they don't need to be completed immediately. Unlike our application servers, a small amount of jobs in the queue may not be a bad thing.

To get an idea of how many workers you'll *probably* need for any given load, you'll need to use Little's Law again:

$$\text{Number of workers} = \text{Average job execution time} * \text{Number of jobs enqueued/sec}$$

Take one bite at a time

Jobs should be as small as possible - not only in terms of lines of code, but in terms of execution time. You'll need the least amount of workers if your jobs reliably execute quickly. Sometimes, of course, that isn't possible - you can't break the transcoding of a video into bite-size chunks, for example.

This gives us a good guideline as to when to split work into different queues - every job in a queue should have an average execution time that's roughly the same as every other job in the queue.

Consider, for example, a background job processor with a single queue. In that queue are 10 video transcoding jobs that take 10 seconds each, and 100 email sending jobs that take 100 milliseconds each. You have 2 workers. If those two workers are *both* processing transcoding jobs, by some trick of the queue ordering, your email sending jobs will have to wait. That's not great - far better here would be to use 2 queues and 1 worker on each queue.

Realize also that background job processors really aren't that different from a distributed map/reduce when you get down to it. A job doesn't *have to* completely finish all the work required - it can place an intermediate work product back on the queue to be finished by another worker.

For example, let's say you need to generate a PDF report. The report contains some statistics from each of your 100 customers. Rather than write this as a single job, write it as three:

- Job 1 should take a Customer object and generate the statistics required for the

report. Once those stats are generated, they should be enqueued as arguments for Job 2.

- Job 2 should take a set of Customer statistics and *reduce* them into a single Report object (perhaps represented as JSON or something like that). Once Job 2 has all the Customer statistics it needs (perhaps it checks if its Report object has a row for every Customer or checks the queue for any incomplete Job 1s), it enqueues its Report object as an argument for Job 3.
- Job 3 takes the completed Report representation and turns it into a PDF document.

Say Job 1 takes about 1 second per job. Since 100 of these jobs are enqueued at once, we can complete this step entirely in $(100 / \text{number of workers})$ seconds. If this was done serially in a single job, it would take 100 seconds every time.

Set timeouts aggressively

Since you're a good programmer and you wrote all of your jobs so that they can be retried idempotently, there's no reason *not* to set any and all network timeouts unusually low. Why?

Network timeouts and long external service responses are an unusual event. We definitely want to make sure jobs can't hang up completely - that would take down an entire worker! But if retrying a job has no drawbacks, then there's really no reason not to set your timeouts aggressively and "call again later" when service conditions are poor.

[The Timeout module is pretty unreliable](#) - where possible, use timeouts built into the libraries you're using.

Job uniqueness is a loser's game

There's not a great way to ensure "uniqueness" for any given background job. [Sidekiq's Enterprise-only unique feature](#) advises that it can only guarantee uniqueness for a limited amount of time.

If you've designed your jobs to be idempotent, you don't care about uniqueness - any given job with any given set of inputs can be executed an infinite number of times with no change in output. Rather than try to use a built-in uniqueness solution, it's far better to implement jobs in an idempotent way.

Sometimes you can accomplish what you want by using throttling instead - for example, you may be trying to query an external service once every 15 minutes and no faster, for example. Throttling is much easier and more reliable to implement - you should be able

to find a solution for your chosen background processor.

What happens when something goes wrong?

Every job should have some kind of failure handler - be sure to ask yourself, what happens if any given line in this worker raises an exception or otherwise goes wrong?

Frequently, you'll probably want to wrap any work inside of either a lock or a database transaction. Transactions ensure that if an exception is raised during your job, any database updates will be rolled back. Make sure there's no way that your job can leave work incomplete - it should either fail completely and do nothing, or succeed and all work should be done. In-between failures can cause data corruption and unexpected behavior.

Set up a red flag

Often, background jobs will fail in such a way that they'll probably *never* complete successfully. You need to be informed when this happens.

As an example, Sidekiq has a "retry" queue. Each time your job fails, Sidekiq places it in the retry queue and tries again according to an exponential backoff formula. After about ~5 retries or so, you can be pretty much certain something's badly wrong. However, by default, Sidekiq will retry your job 25 times before moving it into its "dead" queue. Many (even most) jobs should raise their "red flag" far earlier than that! So be conscious of how your "retry" options are being set, and make sure jobs raise their hands and ask for assistance as soon as it's clear they're failing without hope of success.

Make sure you're using an exception notifier service, and configure it so that you're notified when a certain number of failures occur. You won't want to be notified when *any* failure occurs, of course: jobs fail all the time in background jobs and are immediately (or nearly so) retried and succeed. Only you, the developer, will understand what a truly *exceptional* failure is for your background job setup to maximize the signal/noise ratio in your exception notifier.

Be smart about memory usage

MRI Ruby, once it's obtained memory from the operating system, only releases it back *very slowly* (over the course of hours). This leads to a fairly common problem with background jobs:

1. A background job uses a large amount of memory to do something - for example, it loads up 1000 ActiveRecord objects into an array with an unlimited `where` query. The Ruby process' memory usage balloons by 100 to 200MB or more.
2. Ruby garbage collects that worker's objects after it has finished, *however*, it does not release that memory back to the operating system.
3. You now have a worker process that appears to be bloated and using 100s of MB of memory, when really it's probably using far less.

Be conscious of the memory usage of your jobs if memory bloat is a problem for you. For example, instead of loading 1000s of ActiveRecord objects at once, [use batch limits with AR's `find_each` method](#), so that no more than X amount of records are loaded at any one time.

Understand your reliability requirements

Depending on what you use your background job processor for, you may have stringent or loose reliability requirements. Ask yourself - Is it important that any of these jobs execute 100% of the time? Background job processors are probably reliable 99.99% of the time (depends on your datastore uptime and other factors), but what happens if any given job just never succeeds? This is probably acceptable for many organizations, but for others it won't be.

Here are some typical problems that can cause a job not to be executed:

- Autoscaling. If you kill a worker while it is processing a job, that job may *not* be returned to the job queue, especially if it is a long-running job and your autoscaler terminates the worker process immediately.
- "Unplugging something from the wall". If any of the parts of your job processor - the datastore, a job worker, or the job scheduler/server - suffer some kind of catastrophic failure, jobs in progress may not be returned to the queue where they belong.

Some NoSQL-backed job processors, like Sidekiq, [have additional "extra reliable" modes](#). Sidekiq Pro uses a "blocking pop/push" operation to ensure that if a worker crashes, the job is still returned to the queue. However, because of the way Redis is implemented, this can put a huge performance tax on the amount of calls Sidekiq needs to complete a job.

In general, reliability and performance are a tradeoff with background job processors. Highly reliable solutions are slow, and extremely fast solutions are not 100% reliable. Also, in general, SQL-database-backed queues are more reliable than NoSQL -backed

queues (and suffer a reduction in performance as a result). Their locking mechanisms are more advanced and can generally provide some degree of [ACID guarantees](#), which almost all NoSQL datastores cannot.

Your queue backend should be in the same datacenter as your app

Nearly every background job processor will require an external datastore - Redis, Postgres, etc. Make sure that datastore is physically located, ideally colocated, with whatever server is running your job processor. If your worker server is in Virginia but your datastore is in California, you're going to be imposing *at least* 50-80 milliseconds of network latency to every job. That's a guaranteed way to slow down your job processing!

To get an idea of just how important this penalty can be, especially in scenarios where the datastore is accessed often (many small jobs), take a look at my caching article and the benchmarks therein. The speed and throughput of a cache varies *greatly* when I use a cache store ~20ms away in the cloud versus when I use one locally on the same machine. The datastore for your background job processor works exactly the same way.

Background Job Processors

Here's a quick overview of the choices available to you in Ruby-land:

Resque

The old standby. [Resque](#) was *the* background job processor in its heyday, and many job processors now support a "Resque-compatible" interface as a result. Resque uses Redis as a datastore. It's got a lot of features and a huge community ecosystem.

However, the project has pretty much stalled in recent years. The last release was two years ago in 2014.

Sidekiq

Sidekiq, four years old now, has come a long way. With a Resque-compatible interface, Sidekiq has quickly become the "job processor of first choice" for most projects. [Sidekiq](#) tends to deal better with high loads than Resque because of its multithreaded architecture - each Sidekiq worker can do the work of 20-25 Resque workers when the work is IO-heavy.

Sidekiq is also backed by Redis and has several additional features available with a paid license.

Sneakers

[Sneakers](#) uses RabbitMQ.

An advantage is that RabbitMQ is persistent, placing it somewhere between a database-backed queue and a more messaging-based queue like Redis. Unlike Redis, RabbitMQ has a mature clustering feature, and queues can even be mirrored across multiple machines, giving you a sort of RAID-like data backup.

Unfortunately, unless you're already using RabbitMQ in your application, Sneakers is tough to recommend - most Rubyists are unfamiliar with RabbitMQ.

Que

Database backed queues have advantages and disadvantages - they're easy to introspect (just use SQL!) and tend to be highly reliable (the underlying database is ACID-compliant, after all!). However, using them with high volumes is usually difficult because of the high amount of disk space required to store large amounts of jobs and locking starts to get slow and expensive. [Que](#) tries to solve these problems using Postgres' advisory locks. Check it out if you're not going to be doing thousands of jobs per minute and need heavy reliability guarantees. Because Que uses lightweight advisory locks, it tends to perform far better than [DelayedJob](#).

Checklist for Your App

- **Background work when it depends on an external network request, need not be done immediately, or usually takes a long time to complete.**
- **Background jobs should be idempotent - that is, running them twice shouldn't break anything.** If your job does something bad when it gets run twice, it isn't idempotent. Rather than relying on "uniqueness" hacks, use database locks to make sure work only happens when it's supposed to.
- **Background jobs should be small - do one unit of work with a single job.** For example, rather than a single job operating on 10,000 records, you should be using 10,001 jobs: one to enqueue all of the jobs, and 10,000 additional jobs to do the work. Take advantage of the parallelization this affords - you're essentially doing small-scale distributed computing.

- **Set aggressive timeouts.** It's better to fail fast than wait for a background job worker to get a response from a slow host.
- **Background jobs should have failure handlers and raise red flags.** Consider what to do in case of failure - usually "try again" is good enough. If a job fails 30 times though, what happens? You should probably be receiving some kind of notification.
- **Consider a SQL-database-backed queue if you need background job reliability. Use alternative datastores if you need speed.**
- **Make sure external databases are in the same datacenter as your main application servers.** Latency adds up fast. Usually, in the US, everyone is in the Amazon us-east-1 datacenter, but that may not be the case. Use `ping` to double-check.

Caching in Rails

Caching in a Rails app is a little bit like that one friend you sometimes have around for dinner, but should really have around more often. Nearly every Rails app that's serious about performance could use more caching, but most Rails apps eschew it entirely! And yet, intelligent use of caching is usually the only path to achieving fast server response times in Rails - easily speeding up ~250ms response times to 50-100ms.

A quick note on definitions - this lesson will only cover "application"-layer caching. I'm leaving HTTP caching (which is a whole 'nother beast, and not even necessary implemented *in* your application) for another lesson.

Why don't we cache as much as we should?

Developers, by our nature, are different from end-users. We understand a lot about what happens behind the scenes in software and web applications. We know that when a typical webpage loads, a lot of code is run, database queries executed, and sometimes services pinged over HTTP. That takes time. We're used to the idea that when you interact with a computer, it takes a little while for the computer to come back with an answer.

End-users are completely different. Your web application is a magical box. End-users have no idea what happens inside of that box. Especially these days, **end-users expect near-instantaneous response from our magical boxes**. Most end-users wanted whatever they're trying to get out of your web-app *yesterday*.

This rings of a truism. Yet, we never set hard performance requirements in our user stories and product specifications. Even though server response time is easy to measure and target, and we know users want fast webpages, we fail to ever say for a particular site or feature: "This page should return a response within 100ms." As a result, performance often gets thrown to the wayside in favor of the next user story, the next great big feature. Performance debt, like technical debt, mounts quickly. **Performance never really becomes a priority until the app is basically in flames** every time someone makes a new request.

In addition, caching isn't always easy. **Cache expiration especially can be a confusing topic**. Bugs in caching behavior tend to happen at the integration layer, usually the least-tested layer of your application. This makes caching bugs insidious and

difficult to find and reproduce.

To make matters worse, **caching best practices seem to be frequently changing** in the Rails world. Key-based what? Russian mall caching? Or was it doll?

Benefits of Caching

So why cache? The answer is simple. Speed. With Ruby, we don't get speed for free because our language isn't fast to begin with. We have to get speed from *executing less Ruby on each request*. The easiest way to do that is with caching. Do the work once, cache the result, serve the cached result in the future.

But how fast do we need to be, really?

[Guidelines for human-computer interaction have been known since computers were first developed in the 1960s](#). The response-time threshold for a user to feel as if they are *freely navigating* your site, without waiting for the site to load, is 1 second or less. That's not a 1-second *response time*, but 1 second "*to glass*" - 1 second from the instant the user clicked or interacted with the site until that interaction is complete (the DOM finishes painting).

1 second "to-glass" is not a long time. First, figure about 50 milliseconds for network latency (this is on desktop, latency on mobile is a whole other discussion). Then, budget another 150ms for loading your JS and CSS resources, building the render tree and painting. Finally, figure *at least* 250 ms for the execution of all the Javascript you've downloaded, and potentially much more than that if your Javascript has a lot of functions tied to the DOM being ready. So before we're even ready to consider how long the server has to respond, we're already about ~500ms in the hole. **In order to consistently achieve a 1 second to glass webpage, server responses should be kept below 300ms.** For a 100-ms-to-glass webpage, [as covered in another post of mine](#), server responses must be kept at around 25-30ms.

300ms per request is not impossible to achieve without caching on a Rails app, especially if you've been diligent with your SQL queries and use of ActiveRecord. But it's a heck of a lot of easier if you do use caching. Most Rails apps I've seen have at least a half dozen pages in the app that consistently take north of 300ms to respond, and could benefit from some caching. In addition, using heavy frameworks in addition to Rails, like Spree, the popular e-commerce framework, can slow down responses significantly due to all the extra Ruby execution they add to each request. Even popular heavyweight gems, like Devise or ActiveAdmin, add thousands of lines of Ruby to each request cycle.

Of course, there will always be areas in your app where caching can't help - your POST endpoints, for example. If whatever your app does in response to a POST or PUT is extremely complicated, caching probably won't help you. But if that's the case, consider moving the work into a background worker instead (a blog post for another day).

Getting started

First, [Rails' official guide on caching](#) is excellent regarding the technical details of Rails' various caching APIs. If you haven't yet, give that page a full read-through.

Later on in the article, I'm going to discuss the different caching backends available to you as a Rails developer. Each has their advantages and disadvantages - some are slow but offer sharing between hosts and servers, some are fast but can't share the cache at all, not even with other processes. Everyone's needs are different. In short, the default cache store, `ActiveSupport::Cache::FileStore` is OK, but if you're going to follow the techniques used in this guide (especially key-based cache expiration), you need to switch to a different cache store eventually.

As a tip to newcomers to caching, my advice is to **ignore action caching and page caching**. The situations where these two techniques can be used is so narrow that these features were removed from Rails as of 4.0. I recommend instead getting comfortable with fragment caching - which I'll cover in detail now.

Profiling Performance

Reading the Logs

Alright, you've got your cache store set up and you're ready to go. But what to cache?

This is where profiling comes in. Rather than trying to guess "in the dark" what areas of your application are performance hotspots, we're going to fire up a profiling tool to tell us exactly what parts of the page are slow.

My preferred tool for this task is the incredible [rack-mini-profiler](#). `rack-mini-profiler` provides an excellent line-by-line breakdown of where *exactly* all the time goes during a particular server response.

However, we don't even have to use `rack-mini-profiler` or even any other profiling tools if we're too lazy and don't want to - Rails provides a total time for page generation out of the box in the logs. It'll look something like this:


```

Started GET "/" for ::1 at 2015-07-15 11:04:54 -0400
Started GET "/" for ::1 at 2015-07-15 11:04:54 -0400
Processing by TodosController#index as HTML
Processing by TodosController#index as HTML
  · Todo Load (1.5ms) SELECT "todos".* FROM "todos" WHERE "todos"."session_user_id" = $1 ORDER
    BY "todos"."created_at" ASC [["session_user_id", "822a03cef9b0342814bd6ff3cd71bb4d"]]
  · Todo Load (1.5ms) SELECT "todos".* FROM "todos" WHERE "todos"."session_user_id" = $1 ORDER
    BY "todos"."created_at" ASC [["session_user_id", "822a03cef9b0342814bd6ff3cd71bb4d"]]
  · Rendered todos/index.html.erb within layouts/application (4.1ms)
  · Rendered todos/index.html.erb within layouts/application (4.1ms)
  · Cache digest for app/views/layouts/application.html.erb: dd829b30db986508fd4a838dcc7ca17d
  · Cache digest for app/views/layouts/application.html.erb: dd829b30db986508fd4a838dcc7ca17d
  · Read fragment views/head/dd829b30db986508fd4a838dcc7ca17d (0.2ms)
  · Read fragment views/head/dd829b30db986508fd4a838dcc7ca17d (0.2ms)
  · Cache digest for app/views/layouts/application.html.erb: dd829b30db986508fd4a838dcc7ca17d
  · Cache digest for app/views/layouts/application.html.erb: dd829b30db986508fd4a838dcc7ca17d
  · Read fragment views/footer/dd829b30db986508fd4a838dcc7ca17d (0.2ms)
  · Read fragment views/footer/dd829b30db986508fd4a838dcc7ca17d (0.2ms)
Completed 200 OK in 16ms (Views: 11.7ms | ActiveRecord: 1.5ms)
Completed 200 OK in 16ms (Views: 11.7ms | ActiveRecord: 1.5ms)

```

```
Completed 200 OK in 110ms (Views: 65.6ms | ActiveRecord: 19.7ms)
```

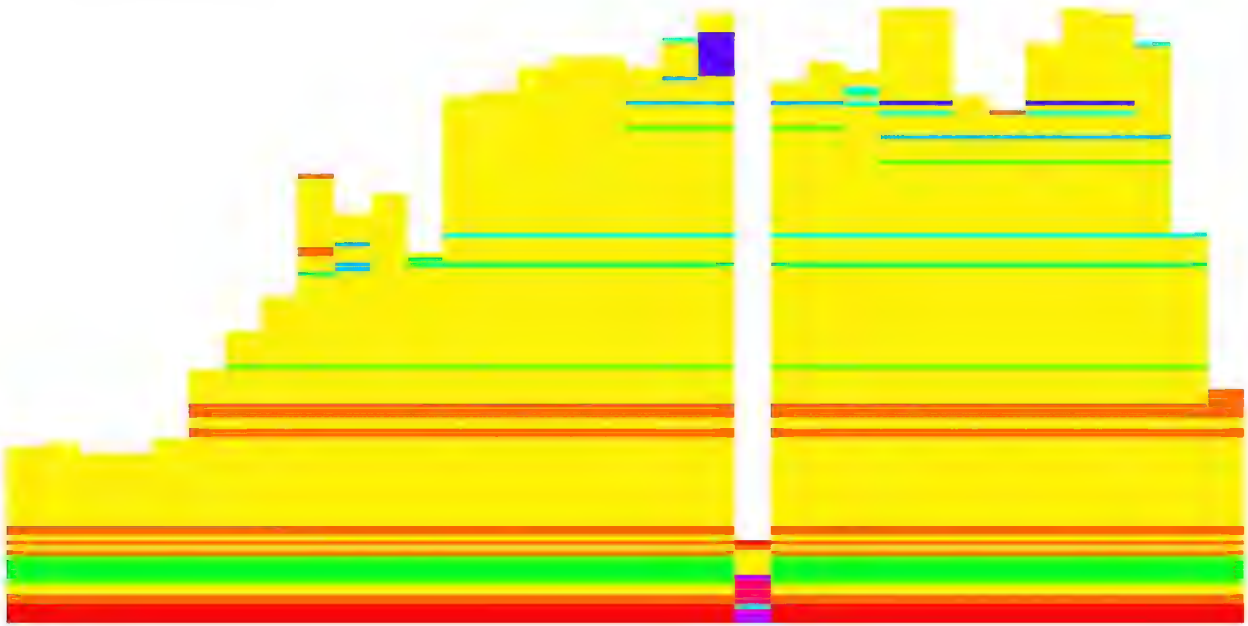
The total time (110ms in this case) is the important one. The amount of time spent in Views is a total of the time spent in your template files (index.html.erb for example). But this can be a little misleading, thanks to how ActiveRecord::Relations lazily loads your data. If you're defining an instance variable with an ActiveRecord::Relation, such as `@users = User.all`, in the controller, but don't do anything with that variable until you start using its results in the view (e.g. `@users.each do ...`), then that query (and reification into ActiveRecord objects), will be counted in the Views number.

ActiveRecord::Relations are *lazily loaded*, meaning the database query isn't executed until the results are actually accessed (usually in your view).

The ActiveRecord number here is also misleading - as far as I can tell from reading the Rails source, this is *not* the amount of time spent executing Ruby in ActiveRecord (building the query, executing the query, and turning the query results into ActiveRecord objects), but only the time spent querying the database (so the actual time spent in DB). Sometimes, especially with complicated queries that use a lot of eager loading, turning the query result into ActiveRecord objects takes a *lot* of time, and that may not be reflected in the ActiveRecord number here.

And where'd the rest of the time go? Rack middleware and controller code mostly. But to get a millisecond-by-millisecond breakdown of *exactly* where your time goes during a request, you'll need `rack-mini-profiler` and the `flamegraph` extension. Using that tool,

you'll be able to see exactly where every millisecond of your time goes during a request on a line-by-line basis.



Production Mode

Whenever I profile Rails apps for performance, **I always do it in production mode**. Not *on* production, of course, but with `RAILS_ENV=production`. Running in production mode ensures that my local environment is close to what the end-user will experience, and also disables code reloading and asset compilation, two things which will massively slow down any Rails request in development mode. Even better if you can use Docker to perfectly mimic the configuration of your production environment. For instance, if you're on Heroku, Heroku recently released some Docker images to help you - but usually virtualization is a mostly unnecessary step in achieving production-like behavior. Mostly, we just need to make sure we're running the Rails server in production mode.

As a quick refresher, here's what you usually have to do to get a Rails app running in production mode on your local machine:

```
export RAILS_ENV=production
rake db:reset
rake assets:precompile
SECRET_KEY_BASE=test rails s
```

In addition, **where security and privacy concerns permit, I always test with a copy of production data**. All too often, database queries in development (like `User.all`) return just 100 or so sample rows, but in production, trigger massive 100,000 row results that

can bring a site crashing to its knees. Either use production data or make your seed data as realistic as possible. This is *especially* important when you're making extensive use of `includes` and Rails' eager loading facilities.

Setting a Goal

Finally, I suggest **setting a maximum average response time, or MART, for your site**. The great thing about performance is that it's usually quite measurable - and what gets measured, gets managed! You may need two MART numbers - one that is achievable in development, with your developer hardware, and one that you use in production, with production hardware.

Unless you have an extremely 1-to-1 production/development setup, using virtualization to control cpu and memory access, you simply will not be able to duplicate performance results across those two environments (though you can come close). That's OK - don't get tripped up by the details. You just need to be sure that your page performance is in the right ballpark.

As an example, let's say we want to build a 100ms-to-glass web app. That requires server response times of 25-50ms. So I'd set my MART in development to be 25ms, and in production, I'd slacken that to about 50ms. My development machine is a little faster than a Heroku dyne (my typical deployment environment), so I give it a little extra time on production.

I'm not aware of any tools yet to do automated testing against your maximum acceptable average response time. We have to do that (for now) manually using benchmarking tools.

Apache Bench

How do we decide what our site's actual average response time is in development? I've only described to you how to read response times from the logs - so is the best way to hit "refresh" in your browser a few times and take your best guess at the average result? Nope.

This is where benchmarking tools like `wrk` and `Apache Bench` come in. `Apache Bench`, or `ab`, is my favorite, so I'll quickly describe how to use it. You can install it on Homebrew with `brew install ab`.

Start your server in production mode, as described earlier. Then fire up `Apache Bench` with the following settings:


```
ab -t 10 -c 10 http://localhost:3000/
```

You'll need to change that URL out as appropriate. The `-t` option controls how long we're going to benchmark for (in seconds), and `-c` controls the number of requests that we'll try at the same time. Set the `-c` option based on your production load - if you have more than an average of 1 request per second (per server), it would be good to increase the `-c` option approximately according to the formula of (Production requests per minute / production servers or dynes) * 2. I usually test with at least `-c 2` to flush out any weird threading/concurrency errors I might have accidentally committed.

Here's some example output from Apache Bench, abridged for clarity:

```
...
Requests per second:    161.04 [#/sec] (mean)
Time per request:      12.419 [ms] (mean)
Time per request:      6.210 [ms] (mean, across all concurrent requests)
...

Percentage of the requests served within a certain time (ms)
 50%      12
 66%      13
 75%      13
 80%      13
 90%      14
 95%      15
 98%      17
 99%      18
100%     21 (longest request)
```

The "time per request" would be the number we compare against our MART. If you also have a 95th percentile goal (95 percent of requests must be faster than X), you can get the comparable time from the chart at the end, next to "95%". Neat, huh?

For a full listing of things you can do with Apache Bench, check out the man page. Notable other options include SSL support, KeepAlive, and POST/PUT support.

Of course, the great thing about this tool is that you can also use it against your production server! If you want to benchmark heavy loads though, it's probably best to run it against your staging environment instead, so that your customers aren't affected!

From here, the workflow is simple - **I don't cache anything unless I'm not meeting my MART**. If my page is slower than my set MART, I dig in with `rack-mini-profiler` to see exactly which parts of the page are slow. In particular, I look for areas where a lot of SQL

is being executed unnecessarily on every request, or where a lot of code is executed repeatedly.

| localhost on Wed, 15 Jul 2015 15:13:24 GMT | | | | |
|--|-------------------|--------------------|----------------------|-----|
| | total time (ms) | from db (ms) | from cache (ms) | |
| GET http://localhost:3000/ | 8.2 | +0.0 | 1 sql | 0.4 |
| Executing action: index | 8.4 | +8.0 | | |
| Rendering: todos/index | 3.5 | +15.0 | 1 sql | 0.5 |
| Rendering: layouts/application | 5.4 | +19.0 | | |
| show time with children | 3.5 | 1.0 | 1 sql | |
| | | | | |
| | connect time (ms) | response time (ms) | load event time (ms) | |
| Connect | 3.0 | +0.0 | | |
| Response | 4.0 | +35.0 | | |
| Load Event | 4.0 | +195.0 | | |
| share | | | | |
| show trivial | | | | |

Caching techniques

Key-based cache expiration

Writing and reading from the cache is pretty easy - again, if you don't know the basics of it, [check out the Rails Guide on this topic](#). **The complicated part of caching is knowing when to expire caches.**

In the old days, Rails developers used to do a lot of manual cache expiration, with Observers and Sweepers. Nowadays, we try to avoid these entirely, and instead use something called *key-based expiration*.

Recall that a cache is simply a collection of keys and values, just like a Hash. In fact, we use hashes as caches all the time in Ruby. Key-based expiration is a cache expiration strategy that expires entries in the cache by making the *cache key* contain information about the *value being cached*, such that when the object changes (in a way that we care about), the cache key for the object also changes. We then leave it to the cache store to expire the (now unused) previous cache key. We never expire entries in the cache manually.

In the case of an ActiveRecord object, we know that every time we change an attribute and save the object to the database, that object's `updated_at` attribute changes. So we can use `updated_at` in our cache keys when caching ActiveRecord objects - each time the ActiveRecord object changes, its `updated_at` changes, busting our cache. Rails knows this and makes it easy for us.

For example, let's say I have a Todo item. I can cache it like this:

```
<% todo = Todo.first %>
<% cache(todo) do %>
  ... a whole lot of work here ...
<% end %>
```

When you give an ActiveRecord object to `cache`, Rails realizes this and generates a cache key that looks a lot like this:

```
views/todos/123-20120806214154/7a1156131a6928cb0026877f8b749ac9
```

The `views` bit is self-explanatory. The `todos` part is based on the Class of the ActiveRecord object. The next bit is a combination of the `id` of the object (123 in this case) and the `updated_at` value (some time in 2012). The final bit is what's called the template tree digest. This is just an md5 hash of the template that this cache key was called in. When the template changes (e.g., you change a line in your template and then push that change to production), your cache busts and regenerates a new cache value. This is super convenient, otherwise we'd have to expire all of our caches by hand when we changed anything in our templates!

Note here that changing anything in the cache key expires the cache. So if any of the following items change for a given Todo item, the cache will expire and new content will be generated:

- The class of the object (unlikely)
- The object's id (also unlikely, since that's the object's primary key)
- The object's `updated_at` attribute (likely, because that changes every time the object is saved)
- Our template changes (possible between deploys)

Note that this technique doesn't *actually* expire any cache keys - it just leaves them unused. Instead of manually expiring entries from the cache, we let the cache itself push out unused values when it begins to run out of space. Or, the cache might use a time-based expiration strategy that expires our old entries after a period of time.

You can give an Array to `cache` and your cache key will be based on a concatenated version of everything in the Array. This is useful for different caches that use the same ActiveRecord objects. Maybe there's a `todo` item view that depends on the `current_user`:

```
<% todo = Todo.first %>
<% cache([current_user, todo]) %>
  ... a whole lot of work here ...
<% end %>
```

Now if the `current_user` gets updated *or* if our `todo` changes, this cache key will expire and be replaced.

Russian Doll Caching

Don't be afraid of the fancy name - the DHH-named caching technique isn't complicated at all.

We all know what Russian dolls look like - one doll contained inside the other. Russian doll caching is just like that - we're going to stack cache fragments inside each other. Let's say we have a list of `Todo` elements:

```
<% cache('todo_list') do %>
  <ul>
    <% @todos.each do |todo| %>
      <% cache(todo) do %>
        <li class="todo"><%= todo.description %></li>
      <% end %>
    <% end %>
  </ul>
<% end %>
```

But there's a problem with my above example code - let's say I change an existing `todo`'s description from "walk the dog" to "feed the cat". When I reload the page, my `todo` list will still show "walk the dog" because, although the inner cache has changed, the outer cache (the one that caches the entire `todo` list) has not! That's not good. We want to re-use the inner fragment caches, but we also want to bust the outer cache at the same time.

Russian doll caching is simply using key-based cache expiration to solve this problem. When the 'inner' cache expires, we also want the outer cache to expire. If the outer cache expires, though, we *don't* want to expire the inner caches. Let's see what that would look like in our `todo_list` example above:

```

<% cache(["todo_list", @todos.map(&:id), @todos.maximum(:updated_at)]) %>
<ul>
  <% @todos.each do |todo| %>
    <% cache(todo) do %>
      <li class="todo"><%= todo.description %></li>
    <% end %>
  <% end %>
</ul>
<% end %>

```

Now, if *any* of the `@todos` change (which will change `@todos.maximum(:updated_at)`) or an `Todo` is deleted or added to `@todos` (changing `@todos.map(&:id)`), our outer cache will be busted. However, any `Todo` items which have not changed will still have the same cache keys in the inner cache, so those cached values will be re-used. Neat, right? That's all there is to it!

In addition, you may have seen the use of the `touch` option on ActiveRecord associations. Calling the `touch` method on an ActiveRecord object updates the record's `updated_at` value in the database. Using this looks like:

```

class Corporation < ActiveRecord::Base
  has_many :cars
end

class Car < ActiveRecord::Base
  belongs_to :corporation, touch: true
end

class Brake < ActiveRecord::Base
  belongs_to :car, touch: true
end

@brake = Brake.first

# calls the touch method on @brake, @brake.car, and @brake.car.corporation.
# @brake.updated_at, @brake.car.updated_at and @brake.car.corporation.updated_at
# will all be equal.
@brake.touch

# changes updated_at on @brake and saves as usual.
# @brake.car and @brake.car.corporation get "touch"ed just like above.
@brake.save

@brake.car.touch # @brake is not touched. @brake.car.corporation is touched.

```

We can use the above behavior to elegantly expire our Russian Doll caches:


```

<% cache @brake.car.corporation %>
  Corporation: <%= @brake.car.corporation.name %>
  <% cache @brake.car %>
    Car: <%= @brake.car.name %>
    <% cache @brake %>
      Brake system: <%= @brake.name %>
    <% end %>
  <% end %>
<% end %>

```

With this cache structure (and the `touch` relationships configured as above), if we call `@brake.car.save`, our two outer caches will expire (because their `updated_at` values changed) but the inner cache (for `@brake`) will be untouched and reused.

Which cache backend should I use?

There are a few options available to Rails developers when choosing a cache backend:

- **ActiveSupport::FileStore** This is the default. With this cache store, all values in the cache are stored on the filesystem.
- **ActiveSupport::MemoryStore** This cache store puts all of the cache values in, essentially, a big thread-safe Hash, effectively storing them in RAM.
- **Memcache and dalli** `dalli` is the most popular client for Memcache cache stores. Memcache was developed for LiveJournal in 2003, and is explicitly designed for web applications.
- **Redis and redis-store** `redis-store` is the most popular client for using Redis as a cache.
- **LRURedux** is a memory-based cache store, like ActiveSupport::MemoryStore, but it was explicitly engineered for performance by Sam Saffron, co-founder of Discourse.

Let's dive into each one one-by-one, comparing some of the advantages and disadvantages of each. At the end, I've prepared some performance benchmarks to give you an idea of some of the performance tradeoffs associated with each cache store.

ActiveSupport::FileStore

FileStore is the default cache implementation for all Rails applications for as far back as I can tell. If you have not explicitly set `config.cache_store` in `production.rb` (or whatever environment), you are using FileStore.

FileStore simply stores all of your cache in a series of files and folders - in `tmp/cache` by default.

Advantages

FileStore works across processes. For example, if I have a single Heroku dyne running a Rails app with Unicorn and I have 3 Unicorn workers, each of those 3 Unicorn workers can share the same cache. So if worker 1 calculates and stores my todolist cache from an earlier example, worker 2 can use that cached value. *However*, this does not work across hosts (since, of course, most hosts don't have access to the same filesystem). Again, on Heroku, while all of the processes on each dyne can share the cache, they cannot share across dynos.

Disk space is cheaper than RAM. Hosted Memcache servers aren't cheap. For example, a 30MB Memcache server will run you a few bucks a month. But a 5GB cache? That'll be \$290/month, please. Ouch. But disk space is a heckuva lot cheaper than RAM, so if you access to a lot of disk space and have a huge cache, FileStore might work well for that.

Disadvantages

Filesystems are slow(ish). Accessing the disk will always be slower than accessing RAM. However, it might be faster than accessing a cache over the network (which we'll get to in a minute).

Caches can't be shared across hosts. Unfortunately, you can't share the cache with any Rails server that doesn't also share your filesystem (across Heroku dynes, for example). This makes FileStore inappropriate for large deployments.

Not an LRU cache. This is FileStore's biggest flaw. FileStore expires entries from the cache based on the *time they were written to the cache*, not *the last time they were recently used/accessed*. This cripples FileStore when dealing with key-based cache expiration. Recall from our examples above that key-based expiration does not actually expire any cache keys manually. When using this technique with FileStore, the cache will simply grow to maximum size (1GB!) and then start expiring cache entries based on the time they were created. If, for example, your todo list was cached first, but is being accessed 10 times per second, FileStore will still expire that item first! Least-Recently-Used cache algorithms (LRU) work much better for key-based cache expiration because they'll expire the entries that haven't been used in a while *first*.

Crashes Heroku dynos Another nail in FileStore's coffin is its complete inadequacy for the ephemeral filesystem of Heroku. Accessing the filesystem is extremely slow on Heroku for this reason, and actually adds to your dynes' "swap memory". I've seen Rails apps slow to a total crawl due to huge FileStore caches on Heroku that take ages to access. In addition, Heroku restarts all dynes every 24 hours. When that happens, the filesystem is reset, wiping your cache!

When should I use ActiveSupport::FileStore?

Reach for FileStore if you have *low request load* (1 or 2 servers) and still need a *very large cache* (>100MB). Also, don't use it on Heroku.

ActiveSupport::MemoryStore

MemoryStore is the other main implementation provided for us by Rails. Instead of storing cached values on the filesystem, MemoryStore stores them directly in RAM in the form of a big Hash.

ActiveSupport::MemoryStore, like all of the other cache stores on this list, is thread-safe.

Advantages

- **It's fast** One of the best-performing caches on my benchmarks (below).
- **It's easy to set up** Simple change `config.cache_store` to `:memory_store`. Tada!

Disadvantages

- **Caches can't be shared across processes or hosts** Unfortunately, the cache cannot be shared across hosts, but it also can't even be shared across processes (for example, Unicorn workers or Puma clustered workers).
- **Caches add to your total RAM usage** Storing data in memory adds to your RAM usage. This is tough on shared environments like Heroku where memory is highly restrained.

When should I use ActiveSupport::MemoryStore?

If you have one or two servers, with a few workers each, and you're storing small amounts of cached data (<20MB), MemoryStore may be right for you.

Memcache and dalli

Memcache is probably the most frequently used and recommended external cache store for Rails apps. Memcache was developed for LiveJournal in 2003, and is used in production by sites like Wordpress.org, Wikipedia, and Youtube.

While Memcache benefits from having some absolutely enormous production deployments, it is under a somewhat slower pace of development than other cache stores (because it's so old and well-used, if it ain't broke, don't fix it).

Advantages

- **Distributed, so all processes and hosts can share** Unlike FileStore and MemoryStore, *all* processes and dynos/hosts share the exact same instance of the cache. We can maximize the benefit of caching because each cache key is only written once across the entire system.

Disadvantages

- **Distributed caches are susceptible to network issues and latency** Of course, it's much, much slower to access a value across the network than it is to access that value in RAM or on the filesystem. Check my benchmarks below for how much of an impact this can have - in some cases, it's extremely substantial.
- **Expensive** Running FileStore or MemoryStore on your own server is free. Usually, you're either going to have to pay to set up your own Memcache instance on AWS or via a service like Memcachier.
- **Cache values are limited to 1MB.** In addition, cache keys are limited to 250 bytes.

When should I use Memcache?

If you're running more than 1-2 hosts, you should be using a distributed cache store. However, I think Redis is a slightly better option, for the reasons I'll outline below.

Redis and redis-store

Redis, like Memcache, is an in-memory, key-value data store. Redis was started in 2009 by Salvatore Sanfilippo, who remains the project lead and sole maintainer today.

In addition to [redis-store](#), there's a new Redis cache gem on the block: [readthis](#). It's under active development and looks promising.

Advantages

- **Distributed, so all processes and hosts can share** Like Memcache, *all* processes and dynos/hosts share the exact same instance of the cache. We can maximize the benefit of caching because each cache key is only written once across the entire system.
- **Allows different eviction policies beyond LRU** Redis allows you to select your own eviction policies, which gives you much more control over what to do when the cache store is full. For a full explanation of how to choose between these policies, check out the [excellent Redis documentation](#).
- **Can persist to disk, allowing hot restarts** Redis can write to disk, unlike Memcache. This allows Redis to write the DB to disk, restart, and then come back up after reloading the persisted DB. No more empty caches after restarting your cache store!

Disadvantages

- **Distributed caches are susceptible to network issues and latency** Of course, it's much, much slower to access a value across the network than it is to access that value in RAM or on the filesystem. Check my benchmarks below for how much of an impact this can have - in some cases, it's extremely substantial.
- **Expensive** Running FileStore or MemoryStore on your own server is free. Usually, you're either going to have to pay to set up your own Redis instance on AWS or via a service like Redis.
- **While Redis supports several data types, redis-store only supports Strings** This is a failure of the `redis-store` gem rather than Redis itself. Redis supports several data types, like Lists, Sets, and Hashes. Memcache, by comparison, only can store Strings. It would be interesting to be able to use the additional data types provided by Redis (which could cut down on a lot of marshaling/serialization).

When should I use Redis?

If you're running more than 2 servers or processes, I recommend using Redis as your cache store.

LRURedux

Developed by Sam Saffron of Discourse, LRURedux is essentially a highly optimized version of ActiveSupport::MemoryStore. Unfortunately, it does not yet provide an ActiveSupport-compatible interface, so you're stuck with using it on a low-level in your app, not as the default Rails cache store for now.

Advantages

- **Ridiculously fast** LRURedux is by far the best-performing cache in my benchmarks.

Disadvantages

- **Caches can't be shared across processes or hosts** Unfortunately, the cache cannot be shared across hosts, but it also can't even be shared across processes (for example, Unicorn workers or Puma clustered workers).
- **Caches add to your total RAM usage** Storing data in memory adds to your RAM usage. This is tough on shared environments like Heroku where memory is highly restrained.
- **Can't use it as a Rails cache store** Yet.

When should I use LRURedux?

Use LRURedux where algorithms require a performant (and large enough to the point where a Hash could grow too large) cache to function.

Cache Benchmarks

Who doesn't love a good benchmark? [All of the benchmark code is available here on GitHub.](#)

Fetch

The most often-used method of all Rails cache stores is `fetch` - if this value exists in the cache, read the value. Otherwise, we write the value by executing the given block. Benchmarking this method tests both read and write performance. `i/s` stands for "iterations/second".

```

LruRedux::ThreadSafeCache:    337353.5 i/s
ActiveSupport::Cache::MemoryStore:    52808.1 i/s - 6.39x slower
ActiveSupport::Cache::FileStore:    12341.5 i/s - 27.33x slower
ActiveSupport::Cache::DalliStore:    6629.1 i/s - 50.89x slower
ActiveSupport::Cache::RedisStore:    6304.6 i/s - 53.51x slower
ActiveSupport::Cache::DalliStore at pub-memcache-13640.us-east-1-1.2.ec2.garantiad
ata.com:13640:    26.9 i/s - 12545.27x slower
ActiveSupport::Cache::RedisStore at pub-redis-11469.us-east-1-4.2.ec2.garantiadata
.com:    25.8 i/s - 13062.87x slower

```

Wow - so here's what we can learn from those results:

- LRURedux, MemoryStore, and FileStore are so fast as to be basically instantaneous.
- Memcache and Redis are still fast when the cache is on the same host.
- When using a host far away across the network, Memcache and Redis suffer significantly, taking about ~50ms per cache read (under extremely heavy load). This means two things - when choosing a Memcache or Redis host, choose the one closest to where your servers are and benchmark its performance. Second, don't cache anything that takes less than ~10-20ms to generate by itself.

Full-stack in a Rails app

For this test, we're going to try caching some content on a webpage in a Rails app. This should give us an idea of how much time read/writing a cache fragment takes when we have to go through the entire request cycle as well.

Essentially, all the app does is set `@cache_key` to a random number between 1 and 16, and then render the following view:

```

<% cache(@cache_key) do %>
  <p><%= SecureRandom.base64(100_000) %></p>
<% end %>

```

Average response time in ms - less is better

The below results were obtained with Apache Bench. The result is the average of 10,000 requests made to a local Rails server in production mode.

- Redis/redis-store (remote) 47.763
- Memcache/Dalli (remote) 43.594

- With caching disabled 10.664
- Memcache/Dalli (localhost) 5.980
- Redis/redis-store (localhost) 5.004
- ActiveSupport::FileStore 4.952
- ActiveSupport::MemoryStore 4.648

Some interesting results here, for sure! Note that the difference between the fastest cache store (MemoryStore) and the uncached version is about 6 milliseconds. We can infer, then, that the amount of work being done by `SecureRandom.base64(100_000)` takes about 6 milliseconds. Accessing the remote cache, in this case, is actually slower than just doing the work!

The lesson? **When using a remote, distributed cache, figure out how long it actually takes to read from the cache.** You can find this out via benchmarking, like I did, or you can even read it from your Rails logs. Make sure you're not caching anything that takes longer to read than it does to write!

Checklist for Your App

- **Use a cache. Understand Rails' caching methods like the back of your hand.** There is no excuse for not using caching in a production application. Any Rails application that cares about performance should be using application-layer caching.
- **Use key-based cache expiration over sweepers or observers.** Anything that manually expires a cache is too much work. Instead, use key-based "Russian Doll" expiration and rely on the cache's "Least-Recently-Used" eviction algorithms.
- **Make sure your cache database is fast to read and write.** Use your logs to make sure that caches are fast. Switch providers until you find one with low latency and fast reads.
- **Consider using an in-memory cache for simple, often-repeated operations.** For certain operations, you may find something like the in-memory `LRURedux` gem to be easier to use.

Lab: Application Caching

Exercise 1

Using rack-mini-profiler as a profiler, identify some areas of Rubygems.org that could use application caching.

If you haven't set up Rubygems.org already, see `RUBYGEMS_SETUP.md`

Implement these changes. The default cache store (the file store) will work fine.

Remember, you'll have to turn on caching in development.

Making Rails Faster Than Sinatra

Summary: Rails has a reputation for being slow among web frameworks - is it? And if so, why? We'll examine Rails' featureset in comparison with other popular Ruby web frameworks, like Lotus, Cuba and Sinatra, by stripping away Rails features until our Rails app is just as fast as a stock Sinatra application.

Rails is slow. We all know that, right? Just have a look at the [Techempower Benchmarks](#) - Rails sits near the bottom. Java and C/C++-powered examples dominate the top of the rankings. Express, a popular framework for Node.js, clocks in orders of magnitude faster. Even Django seems to do better than Rails.

Sidenote: Don't worry, I realize that, as far as benchmarks go, TechEmpower is definitely flawed. This is just for rhetoric's sake. Stay with me here.

Yet, Rails is used successfully at some of the top 1000 websites in the world, as ranked by Alexa. They tend to report respectable median response times, too:

| Website | Alexa Global Ranking | Reported Median Response Time |
|--------------|----------------------|-------------------------------|
| Basecamp.com | #961 | 62 ms (500 req/sec) |
| Shopify | #490 | ~90-100ms, (2000 req/sec) |
| Github | #86 | 45-65 ms |
| Airbnb | #532 | ? |

Sidenote: Since Shopify is public, we can also guess at how much it's spending on servers to support this. [Shopify spent \\$3.2mm on providing subscription services in 2014, for revenue of of \\$38.3mm.](#) Not bad.

So Rails performs poorly on many benchmarks, but seems to have the chops to run Top 1000 (and even Top 100) websites. Why is this? How can Rails perform so poorly in the micro and yet so well in the micro?

Let's create our own micro-benchmark to start getting some answers.

Benchmarking with wrk

There are several benchmarking tools available for measuring how many requests-per-second a web app can handle - for this tutorial, we're going to use `wrk`.

In designing this benchmark, I want to answer the question - how much framework overhead is involved in generating the response to an extremely simple request? The application, in this case, should be extremely simplistic -- we're measuring the speed of the framework rather than the application.

We're just going to render a simple "Hello World!" text response with a 200 status code. I don't even want to introduce JSON into this, because the different frameworks might approach serialization differently. Rendering a "Hello World!" should be the purest way to measure how much work the framework does to render a response.

Here's a repository with the application code I used. To keep all of these frameworks on roughly the same footing, I assured that:

- None of their layout/view features were used - all are rendering plaintext responses.
- All are running in their respective "production" modes if applicable.
- Logging was disabled (at these high loads, logging can take up a significant amount of time).

I ran each application with:

```
RACK_ENV=production puma -q -w 4 -t 16:16 --preload path/to/app
```

and benchmarked that against `wrk` running 100 concurrent connections in 16 threads for 10 seconds. I ran this test three times and took the fastest result.

```
wrk -c 100 -t 16 http://localhost:9292/
```

But who cares about that - we came here to see some numbers, dammit!

| Framework | Requests/sec | Memory usage per worker |
|-----------|--------------|-------------------------|
| Cuba | 8555 | 15 MB |
| Lotus | 5128 | 83 MB |
| Sinatra | 4068 | 22 MB |
| Rails | 1388 | 70 MB |
| Rack | 12158 | 15 MB |

We can immediately identify three tiers, or groupings, of performance here:

- **Rack - Bare Metal™** All of the chosen frameworks here are Rack applications. Really, each one's feature-set *must* be a *strict superset* of whatever Rack is doing. So it's no surprise that Rack is the fastest out of all of these frameworks.
- **Rack Wrappers** Next come the two "Rack Wrapper" frameworks - Cuba and Sinatra. Both are intended to be simple wrappers around Rack, basically making the process of making a Rack-compliant application easier. I was mostly impressed in terms of the memory usage for each of these frameworks - both enjoy almost zero additional overhead over Rack.
- **Rails and Rails-likes** Finally, we have Rails. While Lotus is a newcomer to the scene, it enjoys excellent performance in this test. However, its featureset imposes a large memory penalty, just like Rails' does.

Rails isn't looking good here - nearly an order of magnitude slower than Rack itself, and almost 5x slower than its most feature-comparable competitor, Lotus.

So what is Rails *doing* that is making it so (relatively) slow?

First, I want to reframe that question. In this extremely simple test, Rails is 10x slower *relatively speaking* that Rack. But what does that mean in *absolute* terms? How many milliseconds are we saving with Rack *per response*? Let's rewrite the graph from above in absolute terms:

| Framework | Milliseconds/request | ms/req <i>more</i> than Rack |
|-----------|----------------------|------------------------------|
| Cuba | 9ms | 4ms |
| Lotus | 14ms | 9ms |
| Sinatra | 14ms | 9ms |
| Rails | 33ms | 28ms |
| Rack | 5ms | N/A |

Cuba - 9.03 ms, 4ms Lotus - 14ms, 9ms Sinatra - 14.6 ms, 9ms Rails - 33.17 ms, 28ms Rack - 5.08ms

Rails is imposing about 28ms of framework overhead *per response*. Of course, this is 28ms *on my hardware*, a shitty Macbook Air from 2011. On actual production hardware, with Xeon processors and whatnot, this absolute difference will likely decrease.

Is 28 milliseconds going to make or break your application? Scroll back up to our example response-times-at-webscale table. From a scalability perspective, perhaps it would be interesting for Shopify if they knocked about 25% of the time off each request by eliminating the Rails overhead compared to, say, Cuba.

However, 25 milliseconds is almost a meaningless amount of time for the end-user experience in a web-browser. From a scalability perspective, Shopify could also *scale horizontally* by running more instances of the Rails application rather than attempt to scale by making their application faster (see [my post on Little's Law](#) for how speeding up average response times affects scale). It's not exactly like Shopify is drowning in server costs - their cost-of-revenue for providing their subscription services was about 10% of revenue in 2014. They'd probably rather just throw more servers at the problem rather than rewrite an application to gain 25 milliseconds. Hardware will only get cheaper as time goes on, too - shrinking this framework overhead gap even further.

While there may be some scalability gains in using microframeworks over Rails, we can almost authoritatively say that there aren't any *end-user performance gains*. Most websites take several seconds to even render the page - the bottleneck in most end-user experiences on the web lies in the front-end, not the backend. Here's some back-of-the-napkin estimates:

| Step | Time |
|-------------------------------------|--------------|
| DNS resolution | 150ms |
| TCP connection opening | 150 ms |
| Network request - latency outbound | 50ms |
| Server backend response | 100ms |
| Network request - latency on return | 50ms |
| HTML document parsing/construction | 200ms |
| Total | 700ms |

The framework overhead represents less than 5% of the end-user's experience. Framework overhead in Ruby is, thus, mostly meaningless when discussing end-user performance. Even for API applications, framework overhead represents a tiny fraction of the overall time spent completing a request.

But really, why is it slow?

Rails is slower than other Ruby frameworks for one reason - it runs more Ruby code to make a response. It's as simple as that. We can find out exactly *what* Ruby code it runs by comparing call stack graphs in `ruby-prof`.

`ruby-prof`'s call-stack graphs are conceptually pretty similar to flamegraphs, which you might have seen in Chrome Timeline or `rack-mini-profiler`. To set up `ruby-prof` in this test, I wanted to make sure I inserted its Rack middleware as high in the stack as possible. As an example, here's how I did it with Sinatra:

```
require_relative './app' # contains BenchmarkApp
require 'ruby-prof'
use Rack::RubyProf, :path => './temp'
run BenchmarkApp
```

Looking at Sinatra's call-stack graph gives us a great insight right out of the box - about 30% of the time spent in these simple requests is spent in Rack middleware. Only 70% of the time is spent actually in your application, and almost all of that is in of that in the [Sinatra::Base#call! method](#).

Almost every Ruby web framework uses Rack middleware to provide basic features. Here's a list of some of the ones Sinatra uses by default, pulled from our call-stack graph:

- [Rack::MethodOverride](#) - Allows POST requests to behave like other HTTP methods by setting a special HTTP header. Rails uses this middleware to support PUT and DELETE HTTP methods.
- [Rack::Head](#) - Converts HEAD HTTP methods into GET requests with no body.
- [Rack::Logger](#) - Provides a simple, consistent logger across the entire application.
- [Rack::Protection](#) (including `FrameOptions`, `JsonCsrf`, `PathTraversal`, and `XSSHeader` middlewares) - Protects against certain types of web attacks.

As far as I can tell, each Rack middleware included by default in a Sinatra app is pretty much required for any production application on the public web.

[Rails' default middleware stack is considerably thicker](#). Checking out the call stack on our default Rails application reveals a lot more complexity - I couldn't fit the entire stack onto my screen!

Speeding Up Rails By Doing Less

We can speed up Rails by thinning out our call stack - by making Rails *do less*. Rails includes some obvious things that Sinatra doesn't - an entire ORM, a mailing framework, a background job framework, and more. Let's build our own `mini-rails` - something that's more on feature parity with Sinatra.

Did you know you can launch a Rails application in less than 140 characters?

```
rackup -r action_controller/railtie -b 'run Class.new(Rails::Application){config.secret_key_base=1;}.initialize!'
```

Sidenote: Neato, huh? The `-r` option is the same thing as using `require`. The `-b` option evaluates the string as Ruby code inside a `".ru"` file, like the `config.ru` file in any Rails app. `Class.new(superclass)` just creates a new anonymous class, which is mounted as a Rack app by `run`

Try this in your console. It's a pretty useless application - because there are no routes, it just renders 404s. But it *is* a Rails application.

Here's a working, but non-tweetable Rails application that's more comparable to our Sinatra app:

```
# config.ru
require 'action_controller/railtie'

class BenchmarkApp < Rails::Application
  config.secret_key_base = "test"
  routes.append { root to: "hello#world" }
end

class HelloController < ActionController::Metal
  def world
    self.response_body = "Hello World!"
  end
end

run BenchmarkApp.initialize!
```

| Framework | Requests/sec | Memory usage per worker |
|------------|--------------|-------------------------|
| Sinatra | 4068 | 22 MB |
| Rails | 1388 | 70 MB |
| Tiny Rails | 1248 | 35 MB |

This tiny 1-file Rails app returns the exact same response body as our other generic Rails app generated by `rails new`. While the overall speed is exactly the same, this Rails app needs about half the memory of a standard Rails app. That's interesting - and not inconsiderable savings when multiplied across 4 workers.

It's all in what I'm `require` ing. Or, rather, what I'm **not* `require` ing.

Rails isn't really a single framework - it's actually seven, bundled into one. We need look no further than Rails' gem specification to tell us this:

```
# rails.gemspec
s.add_dependency 'activesupport', version
s.add_dependency 'actionpack',    version
s.add_dependency 'actionview',    version
s.add_dependency 'activemodel',   version
s.add_dependency 'activerecord',   version
s.add_dependency 'actionmailer',   version
s.add_dependency 'activejob',       version
s.add_dependency 'railties',       version
```

Although the Rails gemspec may declare a *dependency* on all of these frameworks, that doesn't necessarily mean all of them are *actually loaded*. `add_dependency` just ensures that Bundler downloads these gems and has them ready to use, we haven't actually `require` d anything yet.

When you type `rails new` and generate a new Rails app, one of the files you'll get is `config/application.rb`. Near the top of that file, you'll see this:

```
require 'rails/all'
```

This is where the Rails gems actually get `require` d. In `railties`, there's an `all.rb` file just like you might expect. It's extremely simple:

```
# rails/railties/lib/rails/all.rb
require "rails"

%w(
  active_record
  action_controller
  action_view
  action_mailer
  active_job
  rails/test_unit
  sprockets
).each do |framework|
  begin
    require "#{framework}/railtie"
  rescue LoadError
  end
end
```

Neat! Already, you should be seeing a possibility for optimization here. Instead of just `require`-ing all of Rails, you should `require` only the parts you need. Here's a brief description of each bit of the framework and the possible memory savings entailed in *not* `require` ing it. All memory numbers were obtained from [derailed_benchmarks](#) .

- Sprockets. 10.3 MB.
- ActiveRecord. 3.5 MB.
- ActionMailer. ~0.5 MB
- ActiveJob. ~0.5 MB

While not requiring parts of Rails we don't need is interesting, in reality, we'll probably only save a couple of MB per instance. These are 100% free gains, though - all we had to do was change a line in our `application.rb` - and free scalability is the best kind! Most common exclusions here would probably be ActionMailer and ActiveJob, followed by ActiveRecord for you folks using MongoDB. If you're an API server, try ripping out Sprockets.

Dropping Down to the Metal :horns:

You may have noticed that the controller I used in this "tiny Rails" app is a little weird.


```
class HelloController < ActionController::Metal
  def world
    self.response_body = "Hello World!"
  end
end
```

First, what's `ActionController::Metal`? Metal is the most basic controller possible in Rails. `ActionController::Base`, what your normal controllers inherit from, literally looks like this:

```
class ActionController::Base < ActionController::Metal
  include SomeModule
  include SomeOtherModule
end
```

`ActionController::Base` includes *a lot* of these modules - [the full list is right here](#). Some of it is stuff you might expect, like helpers, redirects, the `render` method. Other stuff, though, you may not need - like `ForceSSL`, `HttpAuthentication`, and `ImplicitRender`. You can build your own `ActionController::Base` by reading the source and including your own modules, piece by piece.

You'll notice I didn't even use the `render` method - Metal doesn't even include that by default. All I can do is modify the response directly:

```
class HelloController < ActionController::Metal
  def world
    self.headers = { "My-Header" => "Header" }
    self.status = 404
    self.response_body = "Hello World!"
  end
end
```

The Logger

Rails logs a *lot* of information about every request. By default, the production log level is `:info`, which still writes to the log on every request. And, again, Rails will write to the *filesystem* by default, which is pretty slow. First, let's change logging to `STDOUT` (Heroku, for example, already does this):

```
config.logger = Logger.new(STDOUT)
```

But what will really increase our speed is disabling the `INFO` level messages. These normally look like this:

```
I, [2015-12-04T16:03:53.336426 #96456] INFO -- : Started GET "/" for 127.0.0.1 at
2015-12-04 16:03:53 -0500
I, [2015-12-04T16:03:53.336626 #96456] INFO -- : Completed 200 OK in 5ms (Views:
4.9ms)
```

In general, I find these useful. However, if you don't, you may consider increasing the log level to `:warn` or even `:error` or `:fatal`. Doing so provides our first real speed boost in our config tweaking. The reason is simple - logging isn't free!

```
config.log_level = :error
```

Finally, we'll add in some config settings normally set for us in `production.rb`, for the final result:

```
require 'action_controller/railtie'

class BenchmarkApp < Rails::Application
  config.secret_key_base = "test"

  # Usually set for us in production.rb
  config.eager_load = true
  config.cache_classes = true
  config.serve_static_files = false

  config.log_level = :error
  config.logger = Logger.new(STDOUT)

  routes.append { root to: "hello#world" }
end

class HelloController < ActionController::Metal
  def world
    self.response_body = "Hello World!"
  end
end

run BenchmarkApp.initialize!
```

| Framework | Requests/sec | Memory usage per worker |
|-----------------------------|--------------|-------------------------|
| Sinatra | 4068 | 22 MB |
| Rails | 1388 | 70 MB |
| Tiny Rails (logger tweaked) | 3360 | 40 MB |

Rails Isn't Slow - Middleware is Slow

If you run `rake middleware` in the root of any Rails app, you'll see dozens of middlewares listed - even in a fresh new Rails app. Not all of these are necessary for every application. Any middleware, even the default middlewares, can be removed in your app's `application.rb` :

```
config.middleware.delete SomeMiddleware
```

But how do you know which middlewares you can delete safely? Here's a guide:

- **Rack::Sendfile**: This middleware is only needed if you serve files from responses - not likely. This middleware [isn't even supported on Heroku](#) and can be safely deleted if using Heroku.
- **ActionDispatch::Cookies**: Only needed if you use cookies. Usually a good candidate for deletion if you're an API-only app.
- **ActionDispatch::Session::CookieStore** For some reason, setting the `session_store` to nil doesn't remove this middleware. You'll have to `delete` it manually if not using any session store (again, usually this is only for API apps).
- **ActionDispatch::Flash** Don't use flashes? Toss out this middleware.
- **Rack::MethodOverride** You only need this middleware if you're serving HTML content to browsers. API-only apps don't need this - delete it.
- **ActionDispatch::RemoteIp** Reports what Rails thinks is the IP of the client by adding a header to the request. If you're not using a proxy (Heroku without SSL, for example), any client can claim to be any IP by changing the `X-Forwarded-For` header, making this middleware unreliable. If you're in that situation or if you don't use this header, you can safely delete the middleware.
- **ActionDispatch::ShowExceptions** This middleware takes care of sending a pretty 500 page to your user. If this middleware is deleted, 500-level exceptions will receive the correct status code but an empty body response. If all your API clients care about is the status code, you may delete this middleware. HTML-serving apps will want to keep it.

- **ActionDispatch::DebugExceptions** Logs exceptions when they occur. If, for some reason, you don't care about exceptions being logged, you can delete this middleware.
- **ActionDispatch::Callbacks** As far as I can tell, this middleware is never actually used by anything in Rails itself. If your app and none of your included gems use `ActionDispatch::Callbacks`, you can safely remove this middleware.
- **ActionDispatch::RequestId** Adds a header to a request/response that identifies it uniquely. If you don't use this (and don't foresee using it) in your logs, you can remove it. Some web servers have their own version of this feature as well, making Rails' superfluous.
- **ActionDispatch::ParamsParser** Required to use the magic `params` hash in controllers. I guess, if you're extremely narrow and not using this, you can delete it. Not recommended.
- **Rack::ConditionalGet** Used for HTTP caching. 99% of apps should use this - so you probably shouldn't remove it.
- **Rack::ETag** Again, used for HTTP caching.
- **Rack::Head** The HEAD HTTP verb is used to request the headers for a response without the body. If you don't care about HTTP compliance or don't expect any HEAD requests, you can remove this.
- **Rack::Runtime** Adds a `X-Runtime` header to a response. If you don't use this in your logs, you can delete it. As far as I can tell, this should not affect services like New Relic, which tend to use a different header for determining response times.

I should emphasize that before removing any default Rack middleware, you should *read the documentation* for that middleware and *test locally first*. Removing any of these middlewares could cause catastrophic bugs. Proceed with caution.

With that warning out of the way, what if we removed *all* of these middlewares *except* the ones Sinatra uses? What would our framework overhead look like then?

Here's what our final, stripped-down Sinatra-like Rails app looks like:

```

require 'action_controller/railtie'

class BenchmarkApp < Rails::Application
  config.secret_key_base = "test"
  config.log_level = :error
  config.cache_classes = true
  config.serve_static_files = false
  config.eager_load = true
  config.logger = Logger.new(STDOUT)
  config.cache_store = nil

  # Remove middleware that do things Sinatra doesn't (by default)
  config.middleware.delete Rack::Sendfile
  config.middleware.delete ActionDispatch::Cookies
  config.middleware.delete ActionDispatch::Session::CookieStore
  config.middleware.delete ActionDispatch::Flash
  config.middleware.delete ActionDispatch::Callbacks
  config.middleware.delete ActionDispatch::RequestId
  config.middleware.delete Rack::Runtime
  config.middleware.delete ActionDispatch::ShowExceptions
  config.middleware.delete ActionDispatch::DebugExceptions
  config.middleware.delete Rack::ConditionalGet
  config.middleware.delete Rack::ETag

  routes.append { root to: "hello#world" }
end

class HelloController < ActionController::Metal
  HEADERS = {
    'X-Frame-Options' => 'SAMEORIGIN',
    'X-XSS-Protection' => '1; mode=block',
    'X-Content-Type-Options' => 'nosniff',
    'Content-Type' => 'text/html'
  }

  def world
    self.headers = HEADERS
    self.response_body = "Hello World!"
  end
end

BenchmarkApp.initialize!

```

This app benchmarks about 25% faster than stock Sinatra on my machine:

| Framework | Requests/sec | Memory usage per worker |
|------------|--------------|-------------------------|
| Cuba | 8555 | 15 MB |
| Lotus | 5128 | 83 MB |
| Sinatra | 4068 | 22 MB |
| Rails | 1388 | 70 MB |
| Tiny Rails | 5107 | 44 MB |
| Rack | 12158 | 15 MB |

Let's tie up some loose ends:

Why does Rails still use more memory than Sinatra? ActiveSupport, mostly. ActiveSupport has a *lot* of code that gets loaded in no matter how little of it we want to use.

Why is this application so much uglier than a stock Sinatra application? That's a good question, to be honest. Rails isn't really *designed* to be used in this manner, and, to be frank, it's not clear that an application that does so little is even really all that *useful*. This is mainly a difference in philosophy in web frameworks - miniframeworks believe you should be handed nothing and made to bolt on all the appropriate parts yourself, while Rails hands you the keys to a Corvette and just trusts you not to drive into the side of a barn.

Checklist for Your App

- **Instead of requiring rails/all, require the parts of the framework you need.**
You're almost certainly requiring code you don't need.
- **Don't log to disk in production.**
- **If using Rails 5, and running an API server, use `config.api_only`.**
- **Remove middleware you're not using.** It adds up.

Lab: Rails Slimming

This lab requires some extra files. To follow along, download the source code for the course and navigate to this lesson. The source code is available on GitHub (you received an invitation) or on Gumroad (in the ZIP archive).

Exercise 1

An extremely simple Hello World application is included in `lab/app.ru`. Using the material from the lesson, modify it until you can return a Hello World JSON response as quickly as possible.

Exceptions as Flow Control

You may have heard this before: "Don't use exceptions as control flow."

What does that mean?

In Ruby, control flow is most often expressed as an if/else/unless branch:

```
if some_thing
  do_this
else
  other_thing
end
```

Control flow is just a mechanism that controls the path of execution of our program. Do we execute this bit of code, or that bit of code over there? That's control flow. Other methods of control flow in Ruby are `while`, `for`, `case`, `loop`, `break`, and `return`.

When we use *exceptions* as control flow, it often looks like this:

```
begin
  do_some_thing
rescue
  other_thing
end
```

Of course, not *all* `rescue`s are control flow. Really, the only thing separating *control flow* from *exception handling* is that control flow directs the ordinary, every-day execution of our program, and exception handling should only be, well, exceptional.

Using exceptions as control flow occurs when you use `rescue` for every-day occurrences. To pick on someone, I'll use the `stripe-ruby` gem.

Here's what creating a charge looks like with Stripe:

```
Stripe::Charge.create(
  :amount => 400,
  :currency => "usd",
  :source => "tok_17IDJb2eZvKYlo2CIMDCEBEt", # obtained with Stripe.js, this corre
sponds to some credit card #
)
```

If the request succeeds, this returns a `Stripe::Charge` object. If it fails, it raises an exception.

Maybe I just have particularly fat fingers or I'm an exceptionally poor typist, but I mis-type my credit card information *all the time*. I've worked at e-commerce companies before and I know, based on their analytics, that a lot of you do too. Having a credit card declined - whether by typo or by the issuing bank - is not an *exceptional* circumstance.

Why does this distinction *matter*?

Exceptions in Ruby are slow. They're slow in MRI, and they're *extremely* slow in JRuby. Let's do some benchmarking to prove my point - we'll compare an if/else statement with begin/rescue and see how many iterations/sec we can do of each.

```

require 'benchmark/ips'

Customer = Struct.new(:status)

class Charge
  class Declined < RuntimeError; end

  def self.create(opts = {})
    false
  end

  def self.create!(opts = {})
    fail Declined
  end
end

class TestBench
  def fast
    customer = Customer.new

    if Charge.create(amount: 400)
      customer.status = :active
    else
      customer.status = :delinquent
    end
  end

  def slow
    customer = Customer.new
    Charge.create!(amount: 400)
    customer.status = :active
    rescue Charge::Declined
      customer.status = :delinquent
    end
  end
end

test = TestBench.new

Benchmark.ips do |x|
  x.report("if/else") { test.fast }
  x.report("exceptions") { test.slow }
  x.compare!
end

```

IN Ruby 2.2.3, the exception-based flow is 3.44x slower. In JRuby 9.0.4.0, it's **229.53x slower!** Wow! Exceptions used to be extremely slow in MRI Ruby as well, back in the 1.8 days - [Ryan Davis](#) believes it was fixed sometime around 2004. Exceptions in

unexceptional circumstances can impose a major performance penalty - *especially* in JRuby (as of this writing, 2016. I'm sure they're working on it).

Let's look at some more common situations where exceptions are used when faster methods might be available - Rails. Rails has several methods which have exception-raising and non-exception-raising versions:

| Regular method | Same method, raises exceptions on fail |
|----------------|--|
| save | save! |
| find | where OR find_by |
| destroy | destroy! |

Think about it - how often is *not finding a record* an exceptional case? It's pretty easy to find some anti-pattern uses of `find` by searching Github for `rescue`

`ActiveRecord::RecordNotFound` . Here's one I found:

```
def user_loggedin?
  User.find(session[:user_id])
rescue ActiveRecord::RecordNotFound
  false
end
```

...which could be written as:

```
def user_loggedin?
  User.find_by(id: session[:user_id])
end
```

Here's another real-world example found on Github:

```
def set_cart
  @cart = Cart.find(session[:cart_id])
rescue ActiveRecord::RecordNotFound
  @cart = Cart.create
  session[:cart_id] = @cart.id
end
```

You could rewrite this as:

```

@cart = Cart.find_by(id: session[:cart_id])
unless @cart
  @cart = Cart.create
  session[:cart_id] = @cart.id
end

```

What do you do when a 3rd-party library is raising exceptions in everyday operation (like the Stripe example I gave above)? Usually, you're kind of stuck from a performance perspective at least, because as long as the underlying library is raising exceptions, they *have* to be rescued somewhere. You're best off looking for libraries that get the same task done without using exceptions.

One area I find that this happens a lot is HTTP libraries - it's common to raise exceptions for 4XX and 5XX errors. This could be a huge performance drag on, say, a web crawler that may be issuing hundreds of requests per second to possibly unreliable URLs. As far as I know, the only Ruby HTTP library that *doesn't* raise exceptions on 4XX/5XX status codes is [Typhoeus](#).

So is there ever a good time to use exceptions for control-flow-like behavior? Jim Weirich didn't think so. [Here's what he said to Avdi Grimm](#):

Exceptions should not be used for flow control, use throw/catch for that. This reserves exceptions for true failure conditions.

Interesting - let's take a look at `throw` and `catch` for a moment - they're awfully underused in Ruby these days.

```

catch(:done) do
  i = 0
  loop do
    i += 1
    throw :done if i > 100_000
  end
end
finish_up

```

A contrived example, I admit. `throw` and `catch` are basically `raise` and `rescue` without a stack trace - the thing that makes exceptions so expensive in Ruby is all the work required to gather up a stack trace and package it in the exception's `backtrace` method. `throw` doesn't do this - all it does is throw a symbol (I used `:done`, but you

can call it whatever) up the stack to be `catch` ed by a block of the same name. We can also provide a second parameter to `throw` that will be returned by the `catch` block. This lets us speed up our exceptions-as-control-flow example from earlier:

```
require 'benchmark/ips'

Customer = Struct.new(:status)

class Charge
  def self.create(opts = {})
    false
  end

  def self.create!(opts = {})
    throw :failed, :delinquent
  end
end

class TestBench
  def fast
    customer = Customer.new

    if Charge.create(amount: 400)
      customer.status = :active
    else
      customer.status = :delinquent
    end
  end

  def slow
    customer = Customer.new
    customer.status = catch(:failed) do
      Charge.create!(amount: 400)
    end
  end
end

test = TestBench.new

Benchmark.ips do |x|
  x.report("if/else") { test.fast }
  x.report("throw/catch") { test.slow }
  x.compare!
end
```

In JRuby, `throw/catch` is 4.4x slower than an `if/else`, but in MRI Ruby it's just 1.4x slower - nearly the same speed!

TL:DR;

When should you use exceptions?

- **Is this a failure?** `begin` and `rescue` are a little cutesy - did you know `fail` is a synonym in Ruby for `raise` ? When looking at a `raise` in your code, ask yourself - could I write `fail` here instead? If not, is this really an exceptional case?
- **Am I throwing away the exception when I rescue it?** When an exception is raised, do you actually do anything when you rescue it? If not, you may be using exceptions as flow control.
- **Can I use throw/catch here instead?** Throw and catch are a much faster replacement for unwinding the stack in situations that require it.
- **Can I find a different 3rd party library that doesn't raise exceptions?** For example, Typhoeus doesn't raise exceptions on HTTP failures.

Checklist for Your App

- **Eliminate exceptions as flow control in your application.** Most exceptions should trigger a 500 error in your application - if a request that returns a 200 response is raising and rescuing exceptions along the way, you have problems. Use `rack-mini-profiler` 's exception-tracing functions to look for such controller actions.

Webservers and I/O models

Scaling is an intimidating topic. Most blog posts and internet resources around scaling Ruby apps are about scaling Ruby to *tens of thousands of requests per minute*. That's Twitter and Shopify scale. These are interesting - it's good to know the ceiling, how much Ruby can achieve - but not useful for the majority of us out there that have apps bigger than 1 server but less than 100 servers. Where's the "beginner's guide" to scaling? I think the problem is that most people aren't comfortable writing about how big they are until they're *huge*.

Thus, most scaling resources for Ruby application developers are completely inappropriate for their needs. The techniques Twitter used to scale from 10 requests/second to 600 requests/second are not going to be appropriate for getting your app from 10 requests/minute to 1000 requests/minute. Mega-scale has its own unique set of problems - database I/O especially becomes an issue, as your app tends to scale horizontally (across processes and machines) while your database scales vertically (adding CPU and RAM). All of this combines to make scaling a tough topic for most Rails application developers. When do I scale up? When do I scale down?

Since I'm limiting this discussion to 1000 rpm or less, here's what I won't discuss: scaling the DB or other datastores like Memcache or Redis, using a high-performance message queue like RabbitMQ or Kafka, or distributing objects. Also, I'm not going to *tell* you how to get faster response times in this post, although doing so will help you scale.

Also, I won't cover devops or anything beyond your application server (Unicorn, Puma, etc.) First, although it seems shocking to admit, I've spent my entire professional career deploying applications to the Heroku platform. I work for small startups with less than 1000 requests/minute scale. Most of the time, you're the sole developer or one of a handful. For small teams at small scales like this, I think Heroku's payoff is immense. Yes, you can pay perhaps even 50% more on your server bill, but the developer hours it saves screwing with Chef/Ansible/Docker/DevOps Flavor Of The Week pays off big time. I just don't have the experiences to share on scaling custom setups (Docker, Chef, what-have-you) on non-Heroku platforms. Second, when you're running less than 1000 requests/minute, your devops workflow doesn't really need to be specialized all that much. All of the material in this post should apply to all Ruby apps, regardless of devops setup.

As a consultant, I've gotten to see quite a few Rails applications. And most of them are *over-scaled* and *wasting money*.

Heroku's dyno sliders and the many services of AWS make scaling simple, but they also make it easy to scale even when you don't need to. Many Rails developers think that scaling dynos or upping their instance size will make their application faster. Yes, scaling dynos on Heroku will NEVER make your application faster *unless* your app has requests queued and waiting most of the time (explained below). Even PX dynos will only make performance more *consistent*, not *faster*. Changing instance *types* on AWS though (for example, T2 to M4) may change performance characteristics of app instances. When they see that their application is slow, their first reflex is to scale dynos or up their instance sizes (indeed - Heroku support will usually encourage them to do just this! Spend more money, that will solve the problem!). Most of the time though, it doesn't help their problem. Their site is still slow.

As a glossary for this post: *host* refers to a single host machine, virtualized or physical. On Heroku, this is a Dyno. Sometimes people will call this a *server*, but for this post, I want to differentiate between your *host machine* and the *application server* that runs on that machine. A single *host* may run many *app servers*, like Unicorn or Puma. On Heroku, a single host runs a single app server. An *app server* has many *app instances*, which may be separate "worker" processes (like Unicorn) or threads (Puma when running on JRuby in multithreaded). For the purposes of this post, a multi-threaded web server with a single app instance on MRI (like Puma) is not an *app instance* because threads cannot be executed at the same time. Thus, a typical Heroku setup might have 1 host/dyno, with 1 app server (1 Puma master process) with 3-4 app instances (Puma clustered workers).

Scaling increases throughput, not speed. Scaling hosts only speeds up response times if requests are spending time waiting to be served by your application. If there are no requests waiting to be served, scaling only wastes money.

In order to learn about how to scale Ruby apps correctly from 1 to 1000 requests/minute, we're going to need to learn a considerable amount about how your application server and HTTP routing actually works.

I'm going to use Heroku as an example, but many custom devops setups work quite similarly.

Ever wondered exactly what the "routing mesh" was or where requests get queued before being routed to your server? Well, you're about to find out.

How requests get routed to app servers

One of the most important decisions you can make when scaling a Ruby web application is what application server you choose. Most Ruby scaling posts are thus out of date, because the Ruby application server world has changed dramatically in the last 5 years, and most of that whirlwind of change has happened only in the last year. However, to understand the advantages and disadvantages of each application server choice, we're going to have to learn how requests even get routed to your application server in the first place.

Understandably, a lot of developers don't understand how, exactly, requests are routed and queued. It isn't simple. Here's the gist of what most Rails devs already understand about Heroku does it:

- "I think routing changed between Bamboo and Cedar stacks."
- "Didn't RapGenius got pretty screwed over back in the day? I think it was because request queueing was being incorrectly reported."
- "I should use Unicorn. Or, wait, I guess Heroku says I should use Puma now. I don't know why."
- "There's a request queue somewhere. I don't really know where."

Heroku's documentation on HTTP routing is a good start, but it doesn't quite explain the whole picture. For example, it's not immediately obvious *why* Heroku recommends Unicorn or Puma as your application server. It also doesn't really lay out where, exactly, requests get "queued" and which queues are the most important. So let's follow a request from start to finish!

The life of a request

When a request comes in to `yourapp.herokuapp.com`, the first place it stops is a load balancer. These load balancers' job is to make sure the load between Heroku's routers is evenly distributed - so they don't do much other than decide to which router the request should go. The load balancer passes off your request to whichever router it thinks is best (Heroku hasn't publicly discussed how their load balancers work or how the load balancers make this decision).

Now we're at the Heroku router. There are an undisclosed number of Heroku routers, but we can safely assume that the number is pretty large (100+?). The router's job is to *find your application's dynos* and *pass on the request to a dyno*. So after spending about 1-5ms locating your dynos, the router will attempt to connect to a *random dyno* in your

app. Yes, a random one. This is where RapGenius got tripped up a few years ago (back then, Heroku was at best unclear and at worst misleading about how the router chose which dyno to route to). Once Heroku has chosen a random dyno, it will then wait *up to five seconds* for that dyno to accept the request and open a connection. While this request is waiting, it is placed in the router's request queue. However, *each router* has *its own* request queue, and since Heroku hasn't told us how many routers it has, there could be a *huge* number of router queues at any given time for your application. Heroku *will* start throwing away requests from the request queue if it gets too large, and it will also try to quarantine dynos that are not responding (but again, it only does this on an individual router basis, so *every router* on Heroku has to individually quarantine bad dynos). Most of the time, this isn't a big deal - the router is able to connect to a dyno almost immediately, and passes the request to the dyno's open TCP socket. Once connected, the socket on the dyno will accept the connection even if the webserver is busy processing other requests. This is called the "backlog" - we'll get to that in a second.

All of this is *basically* how most custom setups use NGINX. [See this DigitalOcean tutorial](#). Sometimes NGINX plays the role of both load balancer and reverse-proxy in these setups. All of this behavior can be duplicated using custom NGINX setups, though you may want to choose more aggressive settings. NGINX can actually actively send health-check requests to upstream application servers to check if they're alive. Custom NGINX setups tend not to have their own request queues, however.

There are two critical details here for Heroku users: the router will *wait up to 5 seconds for a successful connection to your dyno* and *while it's waiting, other requests will wait in the webserver's backlog*.

Connecting to your server - the importance of server choice

The router (custom setup people - when I say router, you say 'NGINX' or 'Apache') attempting to connect to the server is *the most critical* stage for you to understand, and what happens differs *greatly* depending on your choice of web server. Here's what happens next, depending on your server choice:

Webrick (Rails default)

Webrick is a single-process web server.

It will keep the router's connection open until it has downloaded the entirety of the request from the router. The router will then move on to the next request. Your Webrick server will then take the request, run your application code, and then send back the response to the router. During all of this time, your host is busy and will not accept connections from other routers. If a router attempts to connect to this host while the request is being processed, the router will wait (up to 5 seconds, on Heroku) until the host is ready. The router will not attempt to open other connections to other dynos while it waits. The problems with Webrick are exaggerated with slow requests and uploads.

If someone is trying to upload a 4K HD video of their cat over a 56k modem, you're out of luck - Webrick is going to sit there and wait while that request downloads, and will not do anything in the meantime. Got a mobile user on a 3G phone? Too bad - Webrick is going to sit there and not accept any other requests while it waits for that user's request to slowly and painfully complete.

Webrick can't deal well with slow client requests or slow application responses.

Thin

Thin is an event-driven, single-process web server. There's a way to run multiple Thins on a single host - however, they must all listen on different sockets, rather than a single socket like Unicorn. This makes the setup Heroku-incompatible.

Thin uses EventMachine under the hood (this process is sometimes called *Evented I/O*. It works not unlike Node.js.), which gives you several benefits, in theory. Thin opens a connection with the router and starts accepting parts of the request. Here's the catch though - if suddenly that request slows down or data stops coming in through the socket, Thin will go off and do something else. This provides Thin some protection from *slow clients*, because no matter how slow a client is, Thin can go off and receive other connections from other routers in the meantime. Only when a request is fully downloaded will Thin pass on your request to your application. In fact, Thin will even write large requests (like uploads) to a temporary file on the disk.

Thin is multi-threaded, not multi-process, and threads only run one at a time on MRI. So while actually running your application, your host becomes unavailable (with all the negative consequences outlined under the Webrick section above). Unless you get fancy with your use of EventMachine, too, Thin cannot accept other requests while waiting for I/O in the application code to finish. For example - if your application code POSTs to a payments service for credit card authorization, Thin cannot accept new requests while waiting for that I/O operation to complete *by default*. Essentially you'd

need to modify your application code to send *events* back to Thin's EventMachine reactor loop to tell Thin "Hey, I'm waiting for I/O, go do something else". [Here's more about how that works.](#)

Thin can deal with slow client requests, but it can't deal with slow application responses or application I/O without a whole lot of custom coding.

Unicorn

Unicorn is a single-threaded, multi-process web server.

Unicorn spawns up a number of "worker processes" (app instances), and those processes all sit and listen on a single Unix socket, coordinated by the "master process". When a connection request comes in from a host, it does *not* go to the master process, but instead directly to the Unicorn socket where all of the worker processes are waiting and listening. This is Unicorn's special sauce - no other Ruby web servers (that I know of) use a Unix domain socket as a sort of "worker pool" with no "master process" interference. A worker process (which is only listening on the socket because it isn't processing a request) accepts the request from the socket. It waits on the socket until the request is fully downloaded (setting off alarm bells yet?) and then stops listening on the socket to go process the request. After it's done processing the request and sending a response, it listens on the socket again.

Unicorn is vulnerable to slow clients You can use NGINX in a custom setup to buffer requests to Unicorn, eliminating the slow-client issue. This is exactly what Passenger does, below. While downloading the request off the socket, Unicorn workers cannot accept any new connections, and that worker becomes unavailable. Essentially, you can only serve as many slow requests as you have Unicorn workers. If you have 3 Unicorn workers and 4 slow requests that take 1000ms to download, the fourth request will have to sit and wait while the other requests are processed. This method is sometimes called *multi-process blocking I/O*. In this way, Unicorn can deal with slow application responses (because free workers can still accept connections while another worker process is off working) but not (many) slow client requests. Notice that Unicorn's socket-based model is a form of *intelligent routing*, because only available application instances will accept requests from the socket.

Phusion Passenger 5

Passenger uses a hybrid model of I/O - it uses a multi-process, worker-based structure like Unicorn, however it also includes a buffering reverse proxy.

This is important - it's a bit like running NGINX in front of your application's workers. In addition, if you pay for Passenger Enterprise, you can run multiple app threads on each worker (like Puma, below). To see why Phusion Passenger 5's built-in reverse proxy (a customized NGINX instance written in C++, *not* Ruby) is important, let's walk through a request to Passenger. Instead of a socket, Heroku's router connects to `NGINX` directly and passes off a request to it. This `NGINX` is a specially optimized build, with a whole lot of fancy techniques that make it extremely efficient at serving Ruby web applications. It will download the *entire request* before forwarding it on to the next step - protecting your workers from slow uploads and other slow clients.

Once it has completed downloading the request, `NGINX` forwards the request on to a HelperAgent process, which determines which worker process should handle the request. Passenger 5 can deal with slow application responses (because its HelperAgent will route requests to unused worker processes) *and* slow clients (because it runs its own instance of `NGINX`, which will buffer them).

Puma (threaded only)

Puma, in its default mode of operation, is a multi-threaded, single-process server.

When an application connects to your host, it connects to an EventMachine-like Reactor thread, which takes care of downloading the request, and can asynchronously wait for slow clients to send their entire request (again, just like Thin). When the request is downloaded, the Reactor *spawns a new Thread* that communicates with your application code, and that thread processes your request. You can specify the maximum number of application Threads running at any given time. Again, in this configuration, Puma is multi-threaded, not multi-process, and threads only run one at a time on MRI Ruby. What's special about Puma, however, is that unlike Thin, you don't have to modify your application code to gain the benefits of threading. Puma automatically yields control back to the process when an application thread waits on I/O. If, for example, your application is waiting for an HTTP response from a payments provider, Puma can still accept requests in the Reactor thread or even complete other requests in different application threads. So while Puma can deliver a big performance increase while waiting on I/O operations (like databases and network requests) while actually running your application, your host becomes unavailable during processing, with all the negative consequences outlined under the Webrick section above. Puma (in threaded-only mode) can deal with slow client requests, but it can't deal with slow, CPU-bound application responses.

Puma (clustered)

Puma has a "clustered" mode, where it combines its multi-threaded model with Unicorn's multi-process model.

In clustered mode, Heroku's routers connect to Puma's "master process", which is essentially just the Reactor part of the Puma example above. The master process' Reactor downloads and buffers incoming requests, then passes them to any available Puma worker sitting on a Unix socket (similar to Unicorn). In clustered mode, then, Puma can deal with slow requests (thanks to a separate master process whose responsibility it is to download requests and pass them on) and slow application responses (thanks to spawning multiple workers).

But what does it all mean?

If you've been paying attention so far, you've realized that a scalable Ruby web application needs **slow client protection** in the form of request buffering, and **slow response protection** in the form of some kind of concurrency - either multithreading or multiprocess/forking (preferably both). That only leaves **Puma in clustered mode** and **Phusion Passenger 5** as scalable solutions for Ruby applications on Heroku running MRI/C Ruby. If you're running your own setup, Unicorn with NGINX becomes a viable option.

Each of these web servers make varying claims about their "speed" - I wouldn't get too caught up on it. All of these web servers can handle 1000s of requests per minute, meaning that it takes them less than 1ms to actually handle a request. If Puma is 0.001ms faster than Unicorn, then that's great, but it really doesn't help you much if your Rails application takes 100ms on average to turn around a request. The biggest difference between Ruby application servers is not their speed, but their varying I/O models and characteristics. As I've discussed above, I think that Puma in clustered mode and Phusion Passenger 5 are really the only serious choices for scaling Ruby application because their I/O models deal well with slow clients and slow applications. They have many other differences in features, and Phusion offers enterprise support for Passenger, so to really know which one is right for you, you'll have to do a full feature comparison for yourself.

"Queue time" - what does it mean?

As we've seen through the above explanation, there isn't really a single "request queue". In fact, your application may be interacting with hundreds of "request queues". Here are all the places a request might "queue":

- At the load balancer, Unlikely, as load balancers are tuned to be fast. (~10 load balancer queues?)
- At any of the 100+ Heroku routers. Remember that each router queue is separate (100+ router queues).
- If using a multiprocess server like Unicorn, Puma or Phusion Passenger, queueing at the "master process" or otherwise inside the host (1 queue per host).

So how in the heck does New Relic know how to report queue times?

Well, this is how RapGenius got burned.

In 2013, RapGenius got burned hard when they discovered that Heroku's "intelligent routing" was not intelligent at all - in fact, it was completely random. Essentially, when Heroku was transitioning from Bamboo to Cedar stacks, they *also* changed the load balancer/router infrastructure for *everyone* - Bamboo and Cedar stacks both! So Bamboo stack apps, like RapGenius, were suddenly getting random routing instead of intelligent routing. By intelligent routing, we just mean something better than random. Usually intelligent routing involves actively pinging the upstream application servers to see if they're available to accept a new request. This decreases wait time at the router.

Even worse, Heroku's infrastructure *still reported stats* as if it had intelligent routing (with a *single* request queue, not one-queue-per-router). Heroku would report queue time back to New Relic (in the form of a HTTP header), which New Relic displayed as the "total queue time". However, that header was only reporting the time that particular request spent *in the router queue*, which, if there are 100s of routers, could be extremely low, regardless of load at the host! Imagine - Heroku connects to Unicorn's master socket, and passes a request onto the socket. Now that request spends 500ms on the socket waiting for an application worker to pick it up. Previously, that 500ms would be unnoticed because only router queue time was reported.

Nowadays, New Relic reports queue times based on an HTTP header reported by Heroku called `REQUEST_START`. This header marks the time when Heroku accepted the request at the load balancer. New Relic just subtracts the time that your application worker started processing the request from `REQUEST_START` to get the queue time. So if `REQUEST_START` is exactly 12:00:00 p.m., and your application doesn't start processing the request until 12:00:00.010, New Relic reports that as 10ms of queue time. What's nice about this is that it takes into account the time spent at all levels: time at the load

balancer, time at the Heroku routers, and time spent queueing on your host (whether in Puma's master process, Unicorn's worker socket, or otherwise). Unfortunately, though, this measurement isn't that accurate - New Relic is comparing system clocks *at the millisecond level* of two different machines. Of course, by setting the correct headers on your own NGINX/apache instance, you can get accurate request queueing times with your custom setup.

When do I scale app instances?

Don't scale your application based on response times alone. Your application may be slowing down due to increased time in the request queue, or it may not. If your request queue is empty and you're scaling hosts, you're just wasting money. Check the time spent in the request queue before scaling.

The same applies to worker hosts. Scale them based on the depth of your job queue. If there aren't any jobs waiting to be processed, scaling your worker hosts is pointless. In effect, your worker dynos and web dynos are exactly the same - they both have incoming jobs (requests) that they need to process, and should be scaled based on the number of jobs that are waiting for processing.

NewRelic provides time spent in the request queue, although there are gems that will help you to measure it yourself. If you're not spending a lot of time (>5-10ms of your average server response time) in the request queue, the benefits to scaling are extremely marginal.

Checklist for Your App

- **Use Puma, Unicorn-behind-NGINX or Phusion Passenger as your application server.** The I/O models of these app servers are most suited for Rails applications. If using Unicorn, it must be behind a reverse proxy like NGINX - do not use Unicorn in environments where you do not control the routing, such as Heroku.

Idiomatically Fast Ruby

Ruby is a language where there's more than one way to do things. This is great, because it usually allows for expressive code - natural language often is complex and has meaning that depends on the context, so our code should allow similar flexibility that languages do.

Ruby is unique for providing simple aliases for certain methods - `map` vs `collect`, for example. Many programmers think this is silly - but imagine if we thought the same way about the English language! "Big and large mean the same thing, we should eliminate one of those words and use only one!" What a sad world we'd live in. The truth is that Ruby isn't written for the benefit of the machine - it's written for the benefit of human beings reading it. And so, where appropriate, we can use different ways of writing the same expression to clearly communicate to the *reader* of the code rather than the Ruby *interpreter*.

However, not all Ruby expressions are *exactly* equivalent (that is, they don't trigger *exactly* the same response from the interpreter). Consider:

```
[ :a, :b, :c ].sort_by { rand }
[ :a, :b, :c ].shuffle
```

Each of these lines will return an array of the original elements in a random order - however, one of these methods is *much* faster than the other. Using `shuffle` is almost 6.6x faster with this small array - for an array of 100,000 elements it's 12x slower! JRuby and MRI Ruby perform almost exactly the same in this respect.

In the case of MRI Ruby, the reason is pretty simple if you look at how each method is implemented. Here's `Array#shuffle`:

```
static VALUE
rb_ary_shuffle(int argc, VALUE argv, VALUE ary)
{
    ary = rb_ary_dup(ary);
    rb_ary_shuffle_bang(argc, argv, ary);
    return ary;
}
```

...and here's `sort_by` :

```

static VALUE
rb_ary_sort_by_bang(VALUE ary)
{
    VALUE sorted;

    RETURN_SIZED_ENUMERATOR(ary, 0, 0, ary_enum_length);
    rb_ary_modify(ary);
    sorted = rb_block_call(ary, rb_intern("sort_by"), 0, 0, sort_by_i, 0);
    rb_ary_replace(ary, sorted);
    return ary;
}

```

`rb_block_call` is doing the actual work of your array - which means that for every element in the array, Ruby has to execute `rand`, then place that element in the new, sorted array. By contrast, `shuffle` uses `rb_ary_shuffle_bang` which sorts your array with pure C. JRuby works almost exactly the same way, though [it's Java instead of C](#).

So this lesson is about faster idioms - when you can do something one of many ways, which way is the fastest?

First, a warning and preface: don't optimize your code unless your metrics tell you to do so. None of the following idioms are silver bullets, and none of them will magically make your Rails application average 100ms/response instead of 300. *However*, as you're writing Ruby, I think you should be aware of the tradeoffs you're making when choosing one particular idiom over another. These are things you just want to have "in the back of your mind" while writing Ruby code, and it's also a "checklist" when you come across a hotspot in your Ruby code and you're looking for ways to optimize it.

Of course, we're going to use `benchmark/ips` for comparing approaches. Here's the generic test script:

```

require 'benchmark/ips'

MY_ARRAY = (1..100_000).to_a

Benchmark.ips do |x|
  x.report("sort_by") { MY_ARRAY.sort_by { rand } }
  x.report("shuffle") { MY_ARRAY.shuffle }
  x.compare!
end

```

I'm only going to include the idioms I think are significantly (more than 1.5x) faster than their alternatives on Ruby 2.2.

```
require 'ostruct'

customer = OpenStruct.new
customer.name = "DHH"
customer.active? = true

# 15x faster
customer = {}
customer[:name] = "DHH"

customer.name # "DHH"
customer[:name] # "DHH", 2x faster
```

Unfortunately, OpenStructs are much slower than Hashes. While they're useful for things like test doubles, using them in production code where a Hash would do is not advised - accessing attributes in an OpenStruct is about *twice* as slow as accessing keys in a Hash, and *creating* them is about 15 times slower.

Array#bsearch

```
(1..100_000_000).to_a.find { |number| number > 77_777_777 }
# Over 3,000,000 times faster
(1..100_000_000).to_a.bsearch find { |number| number > 77_777_777 }
```

When working with sorted arrays (for example, Arrays converted from Ranges), using [Array#bsearch](#) is much faster than `Array#find` (alias of `Array#detect`). This speed difference gets wider when the array gets larger - on huge arrays, the difference can be in the magnitude of *5-10 seconds*, or relatively speaking, *3 million times faster*.

Of course, for anyone with a computer science degree, the reason is obvious - binary searching is $O(\log n)$ in the average case, and `Array#find` is a naive implementation that is $O(n)$.

`Array#bsearch` was added with Ruby 2.0.

Arrays vs Sets #include?

Arrays and [Sets](#), while similar, have different performance characteristics.

Recall that Sets are basically Arrays but are unordered and do not have duplicates. These critical differences allow Sets to actually be syntax sugar on top of Hashes. If you think about it, an Array that is unordered with no duplicates is exactly like a Hash with no

loop vs while true

```
loop do
  some_work
  break if some_thing
end

# 4x faster
while true do
  some_work
  break if some_thing
end
```

Starting off with a weird one - `loop` is just slow in Ruby. `while true` is almost 4x faster.

The reason for this is more clear if you look at `loop`'s implementation. `loop` is just fancy syntax for creating an Enumerator that never ends (until it sees a `break`). `while`, meanwhile, is not implemented as a method on Kernel, but is an optimized control structure implemented entirely in C.

Splatting arguments

Splatting arguments is extremely slow in Ruby when passing large amounts of data - check this out:

```
arguments = (1..100).to_a

MyModule.some_method(*arguments)
MyModule.some_method(arguments) # 3x faster
```

The first way of passing `arguments` is almost **3x** slower than just passing the array. Whoa! This just gets worse for larger arrays.

OpenStruct

`OpenStruct`'s are basically fancy Hashes - in fact, they just use a Hash object internally to keep track of their state. Normally we use `OpenStruct`s to have a Hash-like object but with some additional flexibility:

values, only keys.

This means that several operations on Sets are faster than the equivalent Array - `include?` is almost 2x faster. Be sure to check each method with a microbenchmark though - some are also slower (for example, checking intersections/unions/differences between Sets, and iterators like `each`).

Array#sample

```
[*1..100].shuffle.first
[*1..100].sample # 18x faster!
```

`Array#shuffle` is an equivalent to `Array#sample.first` - except it allocates one less Array. *Using `Array#sample.first` is 18x slower than `Array#sample`!*

Enumerable#flat_map

```
ARRAY = (1..100).to_a

ARRAY.map { |e| [e, e] }.flatten
ARRAY.flat_map { |e| [e, e] } # 1.5x faster
```

This is a simple one - `some_array.flatten.map` creates 2 new arrays (one of `some_array` flattened and then one after `map` is run), but `some_array.flat_map` only creates 1 new array. This makes `array.flat_map` about 1.5x faster!

Enumerable#find_index

Same thing here - rather than `some_array.index(some_array.find(element))`, `some_array.find_index(element)` not only reads better, but it's about 1.5x faster!

In this case, rather than creating another array, `find_index` is optimized in C.

#detect vs #select.first, reverse.detect vs select.last

`select.first` and `select.last` are almost always slower than using `detect` - about 4 times slower on an Array of just 100 elements, in fact!

The reason is simple if you think about it - `select` has to iterate over the *entire* array. Consider:

```
(1..100).to_a.select { |el| el == 15 }.first
```

Ruby will execute the block inside `select` 100 times! `find` short_circuits as soon as it finds the correct element:

```
(1..100).to_a.find { |el| el == 15 }
```

This will only execute 15 times - once Ruby finds the *first* element that makes the block `true`, it will stop looking.

Range#cover?

```
NEW_YEAR = Date.new(2015, 1, 1)
NEW_YEAR_EVE = Date.new(2015, 12, 31)
MY_BIRTHDAY = Date.new(2015, 9, 17)

(NEW_YEAR..NEW_YEAR_EVE).include? MY_BIRTHDAY
# 482x faster
(NEW_YEAR..NEW_YEAR_EVE).cover? MY_BIRTHDAY
```

Although they both return the same result, `Range#include?` (and its alias, `Range#member?`) is slower than `Range#cover?` because `#include?` has to iterate over every element in the range. `#cover?` just checks if the argument is between the beginning and end of the range.

This difference is especially apparent for large Ranges of Dates, where `#cover?` can be **500x faster!**

`Range#include?` is just as fast as `Range#cover?`, however, if the Range is numeric. With a numeric Range, `#include?` and `#cover?` use the same implementation.

Hash[]

```
HASH = Hash[*('a'..'z').to_a]

HASH.dup
# 2x faster
Hash[HASH]
```


Here's a weird one - `Hash.dup` is slower than `Hash[]`, almost 2x as much! [@tenderlove discovered this one](#) when tracking down performance issues in Rails integration tests.

Two things to be aware of here - `Hash[]` is considerably less idiomatic, and it doesn't rehash the new object.

Block arguments

```
def some_method &block
  block.call
end

def some_method_without_block_args
  yield
end

some_method { 1 + 1 }
# ~3-4x faster
some_method_without_block_args { 1 + 1 }
```

Block arguments are, unfortunately, slower than just `yield` ing. Up to 4x slower on MRI Ruby, in fact! [Here's a detailed explanation on Omniref](#). In fact, just having a block argument *at all*, even if it isn't called, slows down a method by 3-4x.

String#start_with? and #end_with?

Got any regexes looking to match on the start or end of a String? Well, `String#start_with` and `#end_with` can be considerably faster - around 3-4x faster, actually.

String#tr vs String#gsub

When replacing a few characters in a string (common when, say, replacing spaces with hyphens for URL-ization), `tr` is up to 4x faster than `gsub` (or even `sub`). `gsub` is designed to work with Regexes, while the implementation of `tr` is much simpler because it only works with strings ([although it has some special syntax with similar things to Regexes, like ^](#)).

Learning more

That about covers the biggest "mini-speed-optimizations" you can do in MRI Ruby as of version 2.2.4. To learn more, [check out the fast-ruby project](#). Most of the examples and source material for this lesson was lifted from this awesome community project.

Checklist for Your App

- **Where possible, use faster idioms.** See the entire Idioms lesson for commonly slow code that can be sped up by a significant amount.

ActionController::Live and ActionController::Streaming

This lesson will cover two critically under-used tools for Rails performance -

`ActionController::Live` and `ActionController::Streaming`. In a way, they're "the same, but different." Both modules are about decoupling us from the usual flow of "request and response", although these modules do different things.

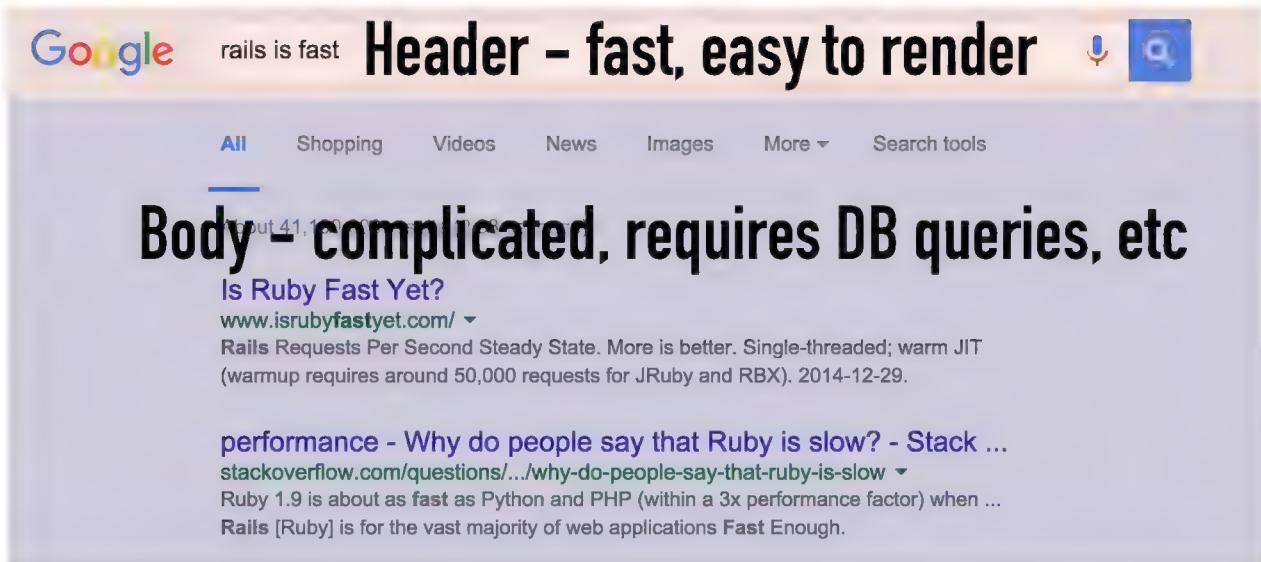
We'll cover each one in turn.

ActionController::Streaming - Start Sending Faster

In a typical request/response cycle, your Rails application will not send data back to the browser until it has completely rendered the response. To put it simply, imagine that once our Rails server has received a request, it takes 100 milliseconds to determine what the "body" string looks like in the final HTTP response. Only when it has completely rendered the entire response does it actually send any data back to the client.

If you think about it, though - does this make a lot of sense? What if, once we had finished rendering the "head" tag, we sent it to the browser right away?

Instead of waiting until the entire response has been rendered, we can start *streaming* the response back to the client as soon as we've determined what the response is. This can actually really help in a number of scenarios, and it's exactly what Google does on its most important page: your search results.



95% of web pages look a lot like Google's search results page. There's a header section, which is simple and comparatively easy to determine what to render. If this was a Rails app, the header would just be a simple form tag and a few images. We can render that in just a few milliseconds!

Don't forget about the `<head>` tag, which is, necessarily, *above* any part of the page `<body>` . These are also usually pretty fast to render on your server.

The main body section, though, is often far more complicated. This is especially true of "search" pages - they often involve one or more big database queries, several partials, and more things that slow down view rendering. 80% of the time in most views is spent in the "body" sections of the page.

As we've discussed before, the *appearance* of speed is just as important to the end user as *actual speed*. We know that, in all cultures, humans typically start reading at the top and work their way down, so we should focus on making the top part of a web-page render as fast as possible, possibly before the eye has even recognized that the rest of the page hasn't loaded yet!

This is exactly Google's strategy in its search results. You can see if a HTTP response is being streamed by looking for the `Transfer-Encoding: chunked` header:

```
$ curl -i www.google.com
HTTP/1.1 200 OK
Date: Tue, 23 Feb 2016 17:49:09 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Transfer-Encoding: chunked
```

Note that there *isn't* a Content-Length header - that's because the server doesn't know how long the response will be yet!

There's another benefit to streaming down responses - a client can start downloading assets, like CSS and JSS, far earlier. Imagine that our homepage usually takes 250 milliseconds to render. Returning a response with streaming enabled might look something like this:

1. Render and stream down the `<head>` tag. Once received, client finds the included `application.css` and `application.js` assets and begins downloading them. If there is any inlined CSS, elements in the (not yet streamed) `body` tag will be immediately rendered using that inlined CSS. Time: 10 milliseconds.
2. Render and stream down the header section of the site. If this is a search page, perhaps it's the usual site header with a few images and a form tag. If there was inlined CSS in the `head`, this tag will be rendered immediately using that CSS, otherwise nothing will render until `application.css` downloads. Time: 10 milliseconds.
3. Begin rendering/streaming body. As ActiveRecord queries begin executing, streaming pauses while they are completed and the corresponding parts of the body are rendered to HTML. Time: 230 milliseconds.

In the above scenario, our browser can start downloading the CSS/JS assets in the `head` 230 milliseconds earlier thanks to response streaming! If enough CSS is inlined, or if the CSS has already been downloaded and is cached from a previous request, we can even start rendering parts of the page about 200 milliseconds earlier than we could without streaming! Neato!

Every browser has supported streaming (sometimes called "flushing") for years - so you don't have to worry about browser support.

Response streaming is also *stupid* easy to use in Rails! Just add `stream: true` to your `render` calls!

```
class PostsController
  def index
    @posts = Post.all
    render stream: true
  end
end
```

There are some caveats though - let's get to those.

Streaming Inverts the Rendering Flow

Streaming inverts the typical rendering flow in a Rails application. Usually, a controller action renders your *view template* first ("app/views/hello/world.html.erb"), then renders the layout ("app/views/layouts/application.html.erb"). Streaming inverts this order. This has some important consequences.

content_for

You may have used `content_for` in your views before.

```
// application.html.erb
<head>
  <script><%= yield :javascript %></script>
</head>
<body> <%= yield %> </body>
// my_view.html.erb

<% content_for :javascript do %>
alert("My view!")
<% end %>
```

This won't work with streaming, though - Rails will render the layout first, and the script tag in the `head` will be empty. Rails pushes the `yield :javascript` part of the response to the client before the `content_for` block in the actual view template is ever executed. Bummer!

In the case above, I would move the `yield :javascript` block to the end of the response:

```
// application.html.erb
<head>
</head>
<body>
<%= yield %>
<script><%= yield :javascript %></script>
</body>
// my_view.html.erb

<% content_for :javascript do %>
alert("My view!")
<% end %>
```

Middleware that modify responses

Streaming doesn't play well with certain middlewares - in particular, it has problems with the ETag middleware.

As an example, here's how the Rack::ETag middleware basically works:

```
def call(env)
  status, headers, body = @app.call(env)

  if should_generate_etag?(status, headers)
    digest = calculate_etag_digest(body)
    headers[ETAG_STRING] = %(W/"#{digest}") if digest
  end

  [status, headers, body]
end
```

Recall from our lesson on HTTP caching that ETags are basically just hash digests of the entire body of a response. If the resource changes, the ETag changes, which tells the browser or client to expire their local cached copy.

Now think about this in the context of streaming - how can you generate the ETag of a response that hasn't been fully generated yet? You can't! ETags are incompatible with streaming for this reason.

Any middleware that attempts to modify the body or headers *after* those responses have been generated is similarly affected.

If you encounter a problem with a middleware, check the Rails issue tracker for the latest.

Partials don't stream

As of writing (February 2016), partials don't stream in quite the way you'd expect. Consider the following template:

```
1

<%= render partial: "sleeps_for_one_second" %>

2

<%= render partial: "sleeps_for_one_second" %>

3
```

...and the partial looks like:

```
Partial rendered!

<% sleep(1) %>

Partial finished!
```

Upon initially loading this page, you might expect to see "1 Partial rendered!" initially upon loading this page, but you won't - instead you'll see "1", wait a second, and then "1 Partial rendered! Partial finished!". When streaming partials, Rails only flushes the output to the stream *after* the entire partial has finished rendering. This isn't really a problem so much as a missed opportunity, but it's something to be aware of.

HAML doesn't work

If you're using HAML for views, you're just straight out of luck - HAML has been incompatible with streaming for years and that doesn't show many signs of changing.

[See their issue tracker for more.](#)

ERB and Slim, my recommended template languages (see "The Easy Mode Stack") are both compatible with streaming.

JSON and XML stream by default

JSON and XML responses are automatically streamed. For example, the following action:

```
def json_endpoint
  big_object = (1..1_000_000).to_a
  render json: big_object
end
```

...will automatically stream. No need for `stream: true`.

Errors are Pretty Clever

What happens if an exception is raised halfway down the page? Rails actually handles this cleverly - if an exception gets raised, Rails appends the following to the response:

```
"><script>window.location = "/500.html"</script></html>
```

This redirects the browser to `500.html` immediately.

Unfortunately, this means streaming *also* breaks the way most exception-catching gems - like Airbrake, Sentry, and Honeybadger - work. These gems may not report exceptions for streaming responses, although supporting streaming is pretty easy. Be sure to test this locally before pushing it to production!

ActionController::Live - Server-Sent Events

You want to build a chat app. Time for WebSockets, right? Fire up that ActionCable and let 'er rip!

Well, not so fast - Rails has an interesting little tool, available since Rails 4, called `ActionController::Live`. It's a bit like one-directional WebSockets that only work from server-to-client, not the other way around. `AC::Live` uses a little-known web API called Server-Sent Events, or SSEs, to establish a long-lived connection between a client and server. Using SSEs, we can send data to the client *without* a corresponding request!

Polling be gone! Rather than polling every five seconds or so, we can simply push events to the browser whenever they happen. In addition, `AC::Live` just uses threads to accomplish its task - unlike ActionCable, there's no need to run a separate server. Neato!

There's one *major* caveat to SSE's - they're not supported by any version of Internet Explorer. However, never fear - there are lot of options for polyfills for IE8+ support if you need them.

Here's what a SSE looks like:

```
id: 1\n
event: chat_message\n
retry: 5000\n
data: "Nate said: SSEs are cool!"\n\n
id: 2\n
event: chat_message\n
retry: 5000\n
data: "Lili said: OMG, I kno rite!"\n\n
```

Note that the three fields of the message are separated by newlines and the entire message is separated by *two* newlines.

The **id** is simply any number, meant to uniquely identify the event. These should probably be incrementing in order to take advantage of SSE's built-in reconnection features - browsers will automatically attempt to reconnect whenever their connection to the server is severed, unlike a WebSockets connection. The browser will send a `Last-Event-ID` header along with this reconnection request, so the server can pick up where it left off and resend any lost events. This field is optional, though.

The **event** field is a generic event name for the data. It can be anything you want, and is generally just for the browser's benefit so you can send multiple types of data along a single SSE stream.

The **retry** field is an integer, in milliseconds, specifying how long the client should wait before attempting to reconnect, if it thinks the connection has been lost. You don't have to specify this field if you don't want - the default behavior is probably fine.

Finally, the **data** is the actual message.

How might we implement a chat application using `ApiController::Live` ?

First, we'll need to make sure we're using Puma or Passenger as our webserver - Unicorn won't work, because it will automatically terminate any connections that are open for more than 30 seconds. Well, that won't work!

The code for a chat application might look something like this:

```

class MessagesController < ApplicationController
  include ActionController::Live

  def stream
    response.headers['Content-Type'] = 'text/event-stream'

    sse = SSE.new(response.stream, retry: 5000, event: "chatMessage")

    begin
      loop do
        Comment.on_change(timeout: 30) do |data|
          sse.write(data)
        end

        sse.write(";")
      end
    rescue IOError
      # connection closed!
    ensure
      sse.close
    end
  end
end

class Message < ActiveRecord::Base
  def self.on_change(opts = {})
    connection.execute "LISTEN #{table_name}"
    loop do
      connection.raw_connection.wait_for_notify(opts[:timeout]) do |event, pid, message|
        yield message
      end
    end
  ensure
    connection.execute "UNLISTEN #{table_name}"
  end

  after_create :notify_new_message

  def notify_new_message
    self.class.connection.execute "NOTIFY #{self.class.table_name}, 'new message'"
  end
end

```

Let's break this down - first we need to set the right content-type for Server-Sent Events. Next, we create a new `SSE` object (provided by `ActionController::Live::SSE`).

Then, we enter a loop - we'll wait for any new Messages using a Postgres LISTEN/NOTIFY pubsub connection. This connection times out every 30 seconds, and then we emit an SSE comment character (",") directly to the stream to make sure our client is still listening. If the heartbeat cannot be delivered, an `IOError` will be raised, causing the connection to be closed.

The reason we send a heartbeat is partly to make sure that the connection is still open, and partly so that no intermediaries close the connection while we're still using it. For example, Heroku kills connections that haven't had data sent over them in the last 55 seconds:

Heroku supports HTTP 1.1 features such as long-polling and streaming responses. An application has an initial 30 second window to respond with a single byte back to the client. However, each byte transmitted thereafter (either received from the client or sent by your application) resets a rolling 55 second window. If no data is sent during the 55 second window, the connection will be terminated.

I've also included some code for what an "on_change" method might look like. I've used Postgres as an example, but you could also use the pub/sub functions of Redis or any other datastore. If I wanted to get *really* fancy I could use some multithreading magic here instead of a database timeout to trigger the heartbeat, but that's definitely beyond the scope of this lesson.

Finally, we're gonna need to listen for new events in the browser. This bit is pretty simple:

```
var source = new EventSource('/messages');
source.addEventListener('chatMessage', function(e) {
  console.log(e.data);
});
```

You'll probably want to do something more useful than writing chat messages to the console, but I think you get the idea.

Checklist for Your App

- **Use streaming liberally with landing pages and complex controller endpoints.** Nearly every large website uses response streaming to improve end-user load times. It's most important to add "render stream: true" on landing pages and

complex actions so that users can start receiving bits of your response as fast as possible, reduce time-to-first-byte and allowing them to download linked assets in the `head` tag as soon as possible. You should also be streaming large file responses, such as large CSV or JSON objects.

- **Use ActionController::Live before trying ActionCable or other "real time" frameworks.** If you don't need "real-time" communication *back* to the server, and only need to push "real-time" updates from server to client, Server Sent Events (SSEs) can be much simpler than using ActionCable.

Action Cable - Friend or Foe?

One of the marquee features of Rails 5 (likely releasing sometime Q1/Q2 2016) is Action Cable, Rails' new framework for dealing with WebSockets. Action Cable has generated a lot of interest, though perhaps for the wrong reasons. "WebSockets are those cool things the Node people get to use, right?" and "I heard WebSockets are The Future™" seem to be the prevailing attitudes, resulting in a lot of confusion and uncertainty about Action Cable's purpose and promise. It doesn't help that current online conversation around WebSockets is thick with overly fancy buzzwords like "realtime" and "full-duplex". In addition, some claim that a WebSockets-based application is somehow more scalable than traditional implementations. What's a Rails application developer to make of all of this?

This won't be a tutorial or a how-to article - instead, we're going to get into the *why* of Action Cable, not the *how*.

Let's start with a review of how we got here - what problem is WebSockets trying to solve? How did we solve this problem in the past?

Don't hit the refresh button!

The Web is built around the HTTP request. In the good old days, you requested a page (GET) and received a response with the page you requested. We developed an extensive methodology (REST) to create a stateless Web based on requesting and modifying resources on the server.

It's important to realize that an HTTP request is *stateless* - in order for us to know *who* is making the request, the request must tell us itself. Without reading the contents of the request, there's really no way of knowing what request belongs to which session. Usually, in Rails, we do this with a secure "signed" cookie that carries a user ID. A signed cookie means that a client can't tamper with its value - important if you want to prevent session hijacking!

As the web grew richer, with video, audio and more replacing the simple text-only pages of yesteryear, we started to crave a constant, uninterrupted connection between server and client. There were places where we wanted the server to communicate back to the client (or vice versa) frequently:

- **Clients needing to send rapidly to the server.** High-throughput environments, like online browser-based games, needed clients and servers to be able to exchange several messages *per second*. Imagine trying to implement a first person shooter's networking code with HTTP requests. Sometimes this is called a "full-duplex" or "bi-directional" communication.
- **"Live" data.** Web pages started to have "live" elements - like a comments section that automatically updated when a new comment was added (without a page refresh), chat rooms, constant-updated stock tickers and the like. We wanted the page to update itself when the data changed on the server *without* user input. Sometimes this is called a "realtime" application, though I find that term buzzwordy and usually inaccurate. "Realtime" implies constant, nano-second resolution updating. The reality is that the comments section on your website probably doesn't change every nano-second. If you're lucky, it'll change once every minute or so. I prefer the term "Live" for this reason. We all know "live" broadcasts are every so slightly delayed by a few seconds, but we'll still call it "live!".
- **Streaming.** HTTP proved unsuitable for streaming data. For many years, streaming video required third-party plugins (remember RealPlayer?). Even now, streaming data other than video remains a complex task without WebSockets (remote desktop connections, for example), and it remains nearly impossible to stream binary data to Javascript without Flash or Java applets (eek!).

The Road to WebSockets

Over the years, we've developed a lot of different solutions to these problems. Some of them haven't really stood the test of time - Flash XMLSocket relays, and `multipart/x-mixed-replace` come to mind. However, several techniques for solving the "realtime" problem(s) are still in use:

Polling

Polling involves the client asking the server, on a set interval (say, three seconds) if there is any new data. Returning to the "live comments" example, let's say we have a page with a comments section. To create this application with polling, we can write some Javascript to ask the server every three seconds for the latest comment data in JSON format. If there is new data, we can update the comment section.

The advantage of polling is that it's rock-solid and extremely simple to set up. For these reasons, it's in wide use all over the Web. It's also resistant to network outage and latency - if you miss 1 or 2 polls because the network went out, for example, no problem! You just keep polling until eventually it works again. Also, thanks to the stateless nature of HTTP, IP address changes (say, a mobile client with data roaming) won't break the application.

However, you might already have alarm bells going off in your head here regarding scalability. You're adding considerable load to your servers by causing every client to hit your server *every* 3 seconds. There are ways to alleviate this - HTTP caching is a good one - but the fact remains, your server will have to return a response to every client every 3 seconds, no matter what.

Also, while polling is acceptable for "live" applications (most people won't notice a 3-second delay in your chat app or comments thread), it isn't appropriate for rapid back-and-forth (like games) or streaming data.

Long-polling

Long-polling is a bit like polling, but without a set interval between requests (or "polls"). The client sends a request to the server for new data - if the server has new data, then it sends a response back like normal. If there isn't any new data, though, it *holds the request open*, effectively creating a persistent connection, and then when it receives new data, completes the response.

Exactly how this is accomplished varies. There are several "sub-techniques" of long-polling you may have heard of, like [BOSH](#) and [Comet](#)). Suffice it so say, long-polling techniques are considerably more complicated than polling, and can often involve weird hacks like hidden iframes.

Long-polling is great when data doesn't change often. Let's say we connect to our live comments, and 45 seconds later a new comment is added. Instead of 15 polls to the server over 45 seconds from a single client, a server would open only 1 persistent connection.

However, it quickly falls apart if data changes often. Instead of a live comments section, consider a stock ticker. A stock's price can change at the millisecond interval (or faster!) during a trading day. That means any time the client asks for new data, the server will return a response immediately. This can get out of hand quickly, because as soon as the

client gets back a response it will make a new request. This could result in 5-10 requests per second *per client*. You would be wise to implement some limits in your client! Then again, as soon as you've done that, your application isn't really RealTime™ anymore!

Server-sent Events (SSEs)

Server-sent Events are essentially a one-way connection from the server to the client. Clients can't use SSEs to send data back to the server. Server-sent Events got turned into a browser API back in 2006, and is currently supported by every major browser *except* any version of Internet Explorer.

Using server-side events is really quite simple from the (Javascript) client's side. You set up an `EventSource` object, define an `onmessage` callback describing what you'll do when you get a new message from the server, and you're off to the races.

Server-sent event support was added to Rails in 4.0, through [ActionController::Live](#).

Serving a client with SSEs requires a persistent connection. This means a few things: using Server-sent events won't work pretty much at all on Heroku, since they'll terminate any connections after 30 seconds. Unicorn will do the same thing, and WEBrick won't work at all. So you're stuck using Puma or Thin, and you can't be on Heroku. Oh, and no one using your site can use Internet Explorer. You can see why ActionController::Live hasn't caught on. It's too bad - the API is really simple and for most implementations ("live" comments, for example) SSE's would work great.

How WebSockets Work

This is the part where I say: "WebSockets to the rescue!" right? Well, maybe. But first, let's investigate what makes them unique.

Persistent, stateful connection

Unlike HTTP requests, WebSocket connections are *stateful*. What does this mean? To use a metaphor - HTTP requests are like a mailbox. All requests come in to the same place, and you have to look at the request (e.g., the return address) to know who sent it to you. In contrast, WebSocket connections are like building a pipe between a server and the client. Instead of all the requests coming in through one place, they're coming in through hundreds of individual pipes. When a new request comes through a pipe, you know *who sent the request*, without even looking at the actual request.

The fact that WebSockets are a *stateful* connection means that the connection between a particular client machine and server must remain constant, otherwise the connection will be broken. For example - a *stateless* protocol like HTTP can be served by any of a dozen or more of your Ruby application's servers, but a WebSocket connection must be maintained by a single instance for the duration of the connection. This is sometimes called "sticky sessions". As far as I can tell, Action Cable solves this problem using Redis. Basically, each Action Cable server instance listens to a Redis pubsub channel. When a new message is published, the Action Cable server rebroadcasts that message to all connected clients. Because all of the Action Cable servers are connected to the same Redis instance, everyone gets the message. It also makes load balancing a lot more difficult. However, in return, you don't need to use cookies or session IDs.

No data frames

To generalize - let's say that every message has *data* and *metadata*. The *data* is the actual thing we're trying to communicate, and *metadata* is data about the data. You might say a communication protocol is more *efficient* if it requires less *metadata* than another protocol.

HTTP needs a decent amount of metadata to work. In HTTP, metadata is carried in the form of HTTP headers.

Here are some sample headers from an HTTP response of a Rails server:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Vary: Accept-Encoding
X-Runtime: 0.121484
X-Powered-By: Phusion Passenger 5.0.14
X-Xss-Protection: 1; mode=block
Set-Cookie: _session_id=f9087b681653d9daf948137f7ece14bf; path=/; secure; HttpOnly
Server: NGINX/1.8.0 + Phusion Passenger 5.0.14
Via: 1.1 vegur
Cache-Control: max-age=0, private, must-revalidate
Date: Wed, 23 Sep 2015 19:43:03 GMT
X-Request-Id: effc7fe2-0ab8-4462-8b64-cb055f5d1b13
Strict-Transport-Security: max-age=31536000
Content-Length: 39095
Connection: close
X-Content-Type-Options: nosniff
Etag: W/"469b11fcecff716247571b85ff1fc7ae"
Status: 200 OK
X-Frame-Options: SAMEORIGIN
```


Yikes, that's 652 bytes before we even get to the data. And we haven't even gotten to the cookie data you sent with the request, which is probably another 2,000 bytes. You can see how inefficient this might be if our data is really small or if we're making a lot of requests.

WebSockets gets rid of most of that. To open a WebSockets connection, the client makes a HTTP request to the server with a special `upgrade` header. The server makes an HTTP response that basically says "Cool, I understand WebSockets, open a WebSockets connection." The client then opens a WebSockets pipe.

Once that WebSockets connection is open, data sent along the pipe requires *hardly any metadata at all*, usually less than about 6 bytes. Neat!

What does all of this mean to us though? Not a whole lot. You could easily do some fancy math here to prove that, since you're eliminating about 2KB of data *per message*, at Google scale you could be saving petabytes of bandwidth. Honestly, I think the savings here are going to vary a lot from application to application, and unless you're at Top 10,000 on Alexa scale, any savings from this might amount to a few bucks on your AWS bill.

Two-way communication

One thing you hear a lot about WebSockets is that they're "full-duplex". What the hell does that mean? Well, clearly, *full duplex* is *better* than *half-duplex* right? That's double the duplexes!

All that full-duplex really means is **simultaneous communication**. With HTTP, the client usually has to complete their request to the server before the server can respond. Not so with WebSockets - clients (and servers) can send messages across the pipe at any time.

The benefits of this to application developers are, in my opinion, somewhat unclear. Polling can simulate full-duplex communication (at a ~3 second resolution, for example) fairly simply. It does reduce latency in certain situations - for example, instead of requiring a request to pass a message back to the client, the server can just send a message immediately, as soon as it's ready. But the applications where ~1-3 second of latency matters are few - gaming being an obvious exception. Basecamp's chat app, Campfire, used 3-second polling for 10 years.

Can i use it?

What browsers can you actually use WebSockets in? Pretty much all of them. This is one of WebSockets' biggest advantages over SSE, their nearest competitor.

[caniuse.com](#) puts [WebSockets' global adoption rate at about 85%](#), with the main laggards being Opera Mini and old versions of the Android browser.

Enter Action Cable

Action Cable was [announced at RailsConf 2015 in DHH's keynote](#). He briefly touched on polling - Basecamp's chat application, Campfire, has used a 3-second polling interval for over 10 years. But then, David said:

"If you can make WebSockets even less work than polling, why wouldn't you do it?"

That's a great mission statement for Action Cable, really. If WebSockets were as easy as polling, we'd all be using it. Continuous updates are just simply better than 3-second updates. If we can get continuous updates without paying any cost, then we should do that.

That's our yardstick - is Action Cable as easy (or easier) to use than polling?

API Overview

Action Cable provides the following:

- A "Cable" or "Connection", a single WebSocket connection from client to server. It's worthwhile to note that Action Cable assumes you will only have one WebSocket connection, and you'll send all the data from your application along different...
- "Channels" - basically subdivisions of the "Cable". A single "Cable" connection has many "Channels".
- A "Broadcaster" - Action Cable provides its own server. Yes, you're going to be running another server process now. Essentially, the Action Cable server just uses Redis' pubsub functions to keep track of what's been broadcasted on what cable and to whom.

Action Cable essentially provides just one class, `ActionCable::Channel::Base`. You're expected to subclass it and make your own Cables, just like ActiveRecord models or ActionController.

Here's a full-stack example, straight from the Action Cable source:

```
# app/channels/application_cable/connection.rb
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    # uniquely identify this connection
    identified_by :current_user

    # called when the client first connects
    def connect
      self.current_user = find_verified_user
    end

    protected
    def find_verified_user
      # session isn't accessible here
      if current_user = User.find(cookies.signed[:user_id])
        current_user
      else
        # writes a log and raises an exception
        reject_unauthorized_connection
      end
    end
  end
end

class WebNotificationsChannel < ApplicationCable::Channel
  def subscribed
    # called every time a
    # client-side subscription is initiated
    stream_from "web_notifications_#{current_user.id}"
  end

  def like(data)
    comment = Comment.find(data['comment_id'])
    comment.like(by: current_user)
    comment.save
  end
end

# Somewhere else in your app
ActionCable.server.broadcast \
  "web_notifications_1", { title: 'New things!', body: 'All shit fit for print'
}

# Client-side coffescript which assumes you've already requested the right to send web notifications
@App = {}
App.cable = Cable.createConsumer "ws://cable.example.com"
App.cable.subscriptions.create "WebNotificationsChannel",
  received: (data) ->
    # Called every time we receive data
    new Notification data['title'], body: data['body']
```

```
connected: ->
  # Called every time we connect
like: (data) ->
  @perform 'like', data
```

A couple of things to notice here:

- Note that the channel name "WebNotificationsChannel" is implicit, based on the name of class.
- We can call the public methods of our Channel from the client side code - I've given an example of "liking" a notification.
- `stream_from` basically establishes a connection between the client and a named Redis pubsub queue.
- `ActionCable.server.broadcast` adds a message in a Redis pubsub queue.
- We have to write some new code for looking up the `current_user`. With polling, usually whatever code we already have written works just fine.

Overall, I think the API is pretty slick. We have that Rails-y feel of a Cable's class methods being exposed to the client automatically, the Cable's class name becoming the name of the channel, et cetera.

Yet, this does feel like a lot of code to me. And, in addition, you're going to have to write more JavaScript than what you have above to connect everything together. Not to mention that now we've got a Redis dependency that we didn't have before.

What I didn't show above is some things that Action Cable gives you for free, like a 3-second heartbeat on all connections. If a client can't be contacted, we automatically disconnect, calling the `unsubscribe` callback on our Channel class.

In addition, [the code, as it stands right now](#), is a joy to read. Short, focused classes with well-named and terse methods. In addition, it's extremely well documented. DHH ain't no slouch. It's a fast read too, weighing in at about 850 lines of Ruby and 200 lines of CoffeeScript.

Performance and Scaling

Readers of my blog will know that my main focus is on performance and Ruby app speed. It's been vaguely claimed that WebSockets offers some sort of scaling or performance benefit to polling. That makes some intuitive sense - surely, large sites like Facebook can't make a 3-second polling interval work.

But moving from polling to WebSockets involves a big trade-off. You're trading a high volume of HTTP requests for a high volume of *persistent connections*. And persistent connections, in a virtual machine like MRI that lacks true concurrency, sounds like trouble. Is it?

Persistent connections

Also note that your server must provide at least the same number of database connections as you have workers. The default worker pool is set to 100, so that means you have to make at least that available.

Action Cable's server uses EventMachine and Celluloid under the hood. However, while Action Cable uses a worker pool to send messages to clients, it's just a regular old Rack app and will need to be configured for concurrency in order to accept many incoming concurrent connections.

What do I mean? Let's turn to `thor`, a WebSockets benchmarking tool. It's a bit like `siege` or `wrk` for WebSockets. We're going to open up 1500 connections to an Action Cable server running on Puma (in default mode, Puma will use up to 16 threads), with varying incoming concurrency:

| Simultaneous WebSocket connections | Mean connection time |
|------------------------------------|----------------------|
| 3 | 17ms |
| 30 | 196ms |
| 300 | 1638ms |

As you can see, Action Cable slows linearly in response to more concurrent connections. Allowing Puma to run in clustered mode, with 4 worker processes, improves results slightly:

| Simultaneous WebSocket connections | Mean connection time |
|------------------------------------|----------------------|
| 3 | 9ms |
| 30 | 89ms |
| 300 | 855 ms |

Interestingly, these numbers are slightly better than a [node.js application I found](#), which seemed to completely crumple under higher load. Here are the results against this node.js chat app:

| Simultaneous WebSocket connections | Mean connection time |
|------------------------------------|----------------------|
| 3 | 5ms |
| 30 | 65ms |
| 300 | 3600 ms |

Unfortunately, I can't really come up with a great performance measure for *outbound* messaging. Really, we're going to have to wait to see what happens with Action Cable in the wild to know the full story behind whether or not it will scale. For now, the I/O performance looks at least comparable to Node. That's surprising to me - I honestly didn't expect Puma and Action Cable to deal with this all that well. I suspect it still may come crashing down in environments that are sending many large pieces of data back and forth quickly, but for ordinary apps I think it will scale well. In addition, the use of the Redis pubsub backend lets us scale horizontally the way we're used to.

What other tools are available?

That concludes our look at Action Cable. What alternatives exist for the Rails developer?

Polling

Let's take the example from above - basically pushing "notifications", like "new message!", out to a waiting client web browser. Instead of pushing, we'll have the client basically ask an endpoint for our notification partial every 5 seconds.

```
function webNotificationPoll(url) {
  $.ajax({
    url : url,
    ifModified : true
  }).done(function(response) {
    $('#notifications').html(response);
    // maybe you call some fancy JS here to pop open the notification window, do some animation, whatever.
  });
}

setInterval(webNotificationPoll($('#notifications').data('url'), 5000);
```

Note that we can use HTTP caching here (the `ifModified` option) to simplify our responses if there are no new notifications available for the user.

Our show controller might be as simple as:

```
class WebNotificationsController < ApplicationController
  def show
    @notifications = current_user.notifications.unread.order(:updated_at)

    if stale?(last_modified: @notifications.last.updated_at.utc, etag: @notifications.last.cache_key)
      render :show
    end

    # note that if stale? returns false, this action
    # automatically returns a 304 not modified.
  end
end
```

Seems pretty straightforward to me. Rather than reaching for Action Cable first, in most "live view" situations, I think I'll continue reaching for polling.

MessageBus

[MessageBus](#) is Sam Saffron's messaging gem. Not limited to server-client interaction, you can also use it for server to server communication.

Here's an example from Sam's README:

```
message_id = MessageBus.publish "/channel", "message"

MessageBus.subscribe "/channel" do |msg|
  # block called in a background thread when message is received
end

// in client JS
MessageBus.start(); // call once at startup

// how often do you want the callback to fire in ms
MessageBus.callbackInterval = 5000;
MessageBus.subscribe("/channel", function(data){
  // data shipped from server
});
```

I like the simplicity of the API. On the client side, it doesn't look all that different from stock polling. However, being backed by Redis and allowing for server-to-server messaging means you're gaining a lot in reliability and flexibility.

In a lot of ways, MessageBus feels like "Action Cable without the WebSockets".

MessageBus does not require a separate server process.

Sync

[Sync](#) is a gem for "real-time" partials in Rails. Under the hood, it uses WebSockets via Faye. In a lot of ways, I feel like Sync is the "application layer" to Action Cable's "transport layer".

The API is no more than changing this:

```
<%= render partial: 'user_row', locals: {user: @user} %>
```

to this:

```
<%= sync partial: 'user_row', resource: @user %>
```

But, unfortunately, it isn't that simple. Sync requires that you sprinkle calls throughout your application any time the `@user` is changed. In the controller, this means adding a `sync_update(@user)` to the controller's update action, `sync_destroy(@user)` to the destroy action, etc. "Syncing" outside of controllers is even more of a nightmare.

Sync seems to extend its fingers all through your application, which feels wrong for a feature that's really just an accident of the view layer. Why should my models and background jobs care that my views are updated over WebSockets?

Others

There are several other solutions available.

- **ActionController::Live**. This might work if you're OK with never supporting Internet Explorer.
- **Faye**. Working with Faye directly is probably more low-level than you'll ever actually need.
- **websocket-rails**. While I'd love another alternative for the "WebSockets for Rails!" space, this gem hasn't been updated since the announcement of Action Cable (actually over a year now).

What do we really want?

Overall, I'm left with a question: I know *developers* want to use WebSockets, but what do our *applications* want? Sometimes the furor around WebSockets feels like it's putting the cart before the horse - are we reaching for the latest, coolest technology when polling is *good enough*?

"If you can make WebSockets easier than polling, then why wouldn't you want WebSockets?"

I'm not sure if Action Cable is easier to use than polling (yet). I'll leave that as an exercise to the reader - after all, it's a subjective question. You can determine that for yourself.

But I think providing Rails developers access to WebSockets is a little bit like showing up at a restaurant and, when you order a sandwich, being told to go make it yourself in the back. WebSockets are, fundamentally, a *transportation* layer, not an *application* in themselves.

Let's return to the three use cases for WebSockets I cited above and see how Action Cable performs on each:

- **Clients needing to send rapidly to the server.** Action Cable seems appropriate for this sort of use case. I'm not sure how many people are out there writing browser-based games with Rails, but the amount of access the developer is given to the transport mechanism seems wholly appropriate here.
- **"Live" data** The "live comments" example. I predict this will be, by far, the most common use case for Action Cable. Here, Action Cable feels like overkill. I would have liked to see DHH and team double down on the "view-over-the-wire" strategy espoused by Turbolinks and make Action Cable something more like "live Rails partials over WebSockets". It would have greatly simplified the amount of work required to get a simple example working. I predict that, upon release, a number of gems that build upon Action Cable will be written to fill this gap.
- **Streaming** Honestly, I don't think anyone with a Ruby web server is streaming binary data to their clients. I could be wrong.

In addition, I'm not sure I buy into "WebSockets completely obviates the need for HTTP!" rhetoric. HTTP comes with a lot of goodies, and by moving away from HTTP we'll lose it all. Caching, routing, multiplexing, gzipping and lot more. You *could* reimplement all of these things in Action Cable, but why?

So when *should* a Rails developer be reaching for Action Cable? At this point, I'm not sure. If you're really just trying to accomplish something like a "live view" or "live partial", I think you may either want to wait for someone to write the inevitable gem on top of Action Cable that makes this easier, or just write it yourself. However, for high-throughput situations, where the client is communicating several times per second back to the server, I think Action Cable could be a great fit.

Checklist for Your App

- **If considering ActionCable, look at the alternatives first.** If all you want is a "live partial", consider the chapter on SSEs and streaming or the `message_bus` gem. Polling is easier to implement for most sites, and has a far less complicated backend setup.

Module 4: The Environment

This module is about "everything else" - the remaining bits of the environment, outside from your application's code or the user's browser, that can make a difference in your app's performance.

The most important lesson in this module is on CDNs - content delivery networks are an as essential optimization tool for any website.

All about CDNs

Using a Content-Delivery Network (also known as a CDN) is one of those things that I think everyone should be using in 2016. There's really no reason *not* to use a CDN on a web application in 2016. Using a CDN to deliver assets can greatly improve network performance for your website - especially in parts of the world far distant from your application servers.

They're also extremely easy to use and deploy - for most Rails apps, it's a one-line change in your configuration.

This lesson gets into the details of CDNs - how they work, why they're helpful, how to set one up on your application, and how to choose between the many vendors available.

What's the role of a CDN?

CDNs perform three critical functions for a typical Ruby web application: decreasing network latency for end-users, reducing load and bandwidth, and (sometimes) modifying your static assets to make them even more efficient.

CDNs decrease network latency

This is probably the biggest win of a CDN.

CDNs use what are called "points of presence" - usually called PoPs, to distribute your cached assets all over the world. You can imagine PoPs as small datacenters, owned by the CDN, distributed around the world. A typical CDN will have a few dozen PoPs placed in strategic locations across the globe. As an example, Amazon CloudFront has 15 PoPs in the United States, 10 in Europe, 9 in Asia, 2 in Australia, and 2 in South America.

PoPs work by caching your content geographically close to the end-user. For example - if your server is located in Amazon's US-East datacenter (located in Virginia), but someone from the Netherlands looks at your site, that response will be cached by a PoP closest to that user. If you were using CloudFront, it would probably be cached in Amazon's Amsterdam PoP. Further requests for that asset by *anyone* nearest to the Amsterdam PoP will be served by Amsterdam, *not* Virginia/US-East.

CDNs can only cache responses that are HTTP-cacheable, which means the correct headers must be set.

However, CDNs also perform important network optimization *even when resources are not cached*. This is called an "uncached origin fetch" (the CDN is fetching an uncached resource from the origin server, your application). Your client browsers connections terminate with a nearby server PoP, *not* your single application server. This means that SSL negotiation and TCP connection opening, among other things, can be significantly faster for client browsers. Your CDN can maintain a pool of open connections between their CDN backbone and your origin server, unlike your client browsers, which are probably connecting to you for the first time.

Many CDNs use this nearby termination on both ends of the connection, meaning that traffic will be routed across the CDN's optimized backbone, further reducing latency.

This also means that one of the key factors in choosing a CDN can be the location of its PoPs - if you have an application which is used heavily by European or Asian users, for example, you will definitely be looking at a different set of CDNs than someone who's optimizing for American users. More on that later.

CDNs reduce load on your application

CDNs are basically free (or almost-free) bandwidth. By serving HTTP-cacheable resources from a CDN instead of your own servers, you're simply saving money.

Consider this - Amazon Web Services has two different pricing rates for bandwidth. One is the rate for their cloud storage service S3 and on-demand computing resource EC2, and the other is for their CDN offering, CloudFront. Bandwidth on EC2 and S3 costs 15 cents for the first 10 terabytes a month. The prices for CloudFront are almost exactly half - eight and half cents for the first 10 terabytes. I'm not a business expert, but "cut your costs by half" sounds like a good strategy to me. As a side note, you can cut costs even further on some CDNs by restricting the number of PoPs that you use in their network.

Some CDNs even provide bandwidth *for free* - Cloudflare, for example, does not have bandwidth limits.

In addition, if, god forbid, you're serving asset requests directly from your application - either your webserver, like NGINX or Apache, or your Rails app directly, as is common on Heroku - consider that the time your server spends serving cacheable HTTP assets is "crowding out" time it could spend serving the kinds of requests it *should* be serving.

Asset requests are usually quite fast, it's true - just a few milliseconds each - but consider that most pages require 3-4 static assets, and multiply that by thousands of users, and you can see how serving your own assets can get out of hand.

Every request served by the CDN is money in your pocket - whether it's in reduced bandwidth bills or in taking load off of your application servers.

CDNs can perform modification of your static assets to make them more efficient

Some CDNs go a step further than acting as just a big, geographically convenient intermediate HTTP cache. Certain CDNs - CloudFlare being the most obvious in this category - will modify your content en-route to make it more efficient.

Here's a shortlist of modifications a CDN may perform to your HTTP-cacheable content:

- **Images may be modified in several ways.** It's extremely common for CDNs to modify images before storing them. This isn't a surprise - images comprise a fair portion of overall Internet bandwidth usage, and most sites poorly compress their own images. JPEGs may be re-compressed at a lower quality level or have EXIF information stripped, PNGs may be re-compressed, GIFs may have frames dropped. Some CDNs offer ways to mobile-optimize images, loading blurred "placeholder" images in place of the actual image, and then "lazy loading" the real image once the user scrolls.
- **Responses may be minified and gzip compressed.** If the origin server hasn't already gzipped or minified their CSS, Javascript or HTML, many CDNs will do this-on-the-fly.

CDNs are a cheap-and-easy way to get some benefits of HTTP/2

As mentioned in the HTTP/2 lesson, using a CDN is a cheap-and-easy way to get some of the benefits of HTTP/2 without changing much.

Usually, the majority of a page's download weight is in its static assets. The document itself is, comparatively, not usually heavy. Just a few kilobytes or so. By moving our assets to an HTTP/2 compatible CDN, we can get a lot of the benefits of HTTP/2 - improved bandwidth management, increased parallelism, and header compression - without changing a single line of code in our application.

My recommended CDN setup

I recommend using a CDN with your Rails application set as the origin server.

Uncacheable document responses, like HTML documents, will be served by the Rails application, but *everything else* should be served by the CDN. I do *not* recommend using intermediate steps - such as uploading assets to S3 first - or using a CDN whose origin you do not control (so-called 3rd-party CDNs, typically used for serving popular CSS or Javascript libraries).

The 12-Factor advantage - simplicity!

[The 12 Factor Application](#) is a set of principles for creating easily maintained web applications. One of these principles (called "factors") is called "Dev-Prod Parity": keep development, staging, and production as similar as possible.

I'm just going to quote a short section here, though the whole document is worth reading:

The twelve-factor developer resists the urge to use different backing services between development and production, even when adapters theoretically abstract away any differences in backing services. Differences between backing services mean that tiny incompatibilities crop up, causing code that worked and passed tests in development or staging to fail in production. These types of errors create friction that disincentivizes continuous deployment. The cost of this friction and the subsequent dampening of continuous deployment is extremely high when considered in aggregate over the lifetime of an application.

Using S3 in production and serving assets directly from the server in development is a common pattern in Rails applications. Unfortunately, this requires the maintenance of an entire upload process, and differences in the configuration of S3 versus your Rails application server means that assets may be served quite differently in production than they are in development. **Always using your Rails application as your CDN's origin reduces the different between production and development, making your life easier.**

In addition, this approach means cacheable documents - like JSON responses - are cached exactly the same way as cacheable static assets, like JS and CSS.

Some may be protesting - but you just told me the benefit of a CDN was to prevent my application server from serving static assets or other cacheable resources! Most Rails applications, which have a dozen or so static assets, are best served by this approach.

On the first request of a certain asset, say `application.css`, the CDN will ask your Rails server for the file, and then will *never ask for it again*. Each static asset should only be served *once* to the CDN, which is no big deal at all!

Of course, there will be scenarios where this is not possible. If your application has user-uploaded content, or static assets are somehow generated dynamically, you must use a separate origin, such as Amazon S3, for those assets. If the number of static assets is few but they are requested many times, use your Rails application as an origin server. If there are many assets which may be requested just a few times each, it's probably worth it to offload those assets to a separate origin entirely.

Avoiding Common Mistakes

Here are some common pitfalls in deploying CDNs on web applications.

The CDNJS pipe dream

Although this isn't common in the Rails community, the use of CDNs whose origin is not owned by you is becoming increasingly common. I'll call them "3rd-party CDNs" - sites like CDNjs and BootstrapCDN. Usually, developers use these to add popular libraries such as Bootstrap, React, and others to their pages. I have a couple of problems with this approach:

- **Gzip performance is adversely affected.** Gzip works best on large files. This makes intuitive sense - a compression algorithm works better when it has more bits to work with, more similar chunks of data to compress. Taking 10 Javascript libraries, putting them into separate files, and compressing them will *always* have a larger total file size than concatenating those 10 libraries into one file and compressing the single file.
- **If using HTTP/1, it's always faster to download 1 resource rather than 2.** Often, sites will include more than 1 of these 3rd-party hosted libraries. However, as we know from the front-end module of this course, this will always be slower in HTTP/1.x than downloading those same resources as a single concatenated file. With HTTP/1.x, we have to open new connections to download each of these resources! Far better to concatenate all of these resources together, like the Rails asset pipeline does by default.
- **Most of these frameworks have parts you don't need.** Particularly in the case of Bootstrap, it doesn't make much sense to download the entirety of these common frameworks. Bootstrap, for example, makes it extremely easy to include only the

parts of the framework that you use. Why use the stock version and waste the bits?

- **Frequently combined, leading to domain explosion.** For some reason, sites often tend to combine these 3rd-party CDNs. Doing so just leads to more DNS lookups, SSL negotiations, and TCP connections - slowing down your page load.
- **The caching benefits are a pipe dream.** Many cite, without data, that because other sites sometimes use these 3rd-party CDNs, many users will already have cached versions stored on their device. Unfortunately, this completely ignores both the prevalence of the use of these 3rd party CDNs, but also ignores that user caches are difficult to rely on. Especially on mobile devices, caches are of a limited size, and files can be evicted at any time.

S3 (or your app server) is not a CDN

I've seen a lot of Rails applications that set an S3 bucket as their `asset_host` and leave it at that - scroll up and re-read the benefits of a CDN. You're not getting any of those - S3 is not geographically distributed (all of your assets will be served from whatever zone you're in, again, probably US-East).

It's easy to mark assets as "Do Not Modify!"

As mentioned, CDNs frequently modify responses in-transit - sometimes, though, you may not want this behavior. An example is medical imaging: medical images have strict standards and usually need to be transmitted losslessly, with no modification of the data.

This is easy to accomplish by setting the `no-transform` directive in a response's Cache-Control headers. A header of `Cache-Control: no-transform` instructs any intermediate caches to *not* modify the resource in any way.

An overview of the CDN options available

Different CDNs have different options - primarily, they differ in how many "bells and whistles" they offer and the location of their Points of Presence. The CDNs in this list don't have any real, appreciable difference in uptime (as measured by third parties) or even in bandwidth speed.

There are also some upmarket CDNs, such as Akamai, which I won't cover here. They're designed for sites in the top 10,000 in the world, which I assume are not reading this guide.

Cloudflare

Cloudflare is my preferred choice for small-scale projects (as with my Skylight and NewRelic reviews, I have no relationship with Cloudflare).

Cloudflare's free tier is an incredible gift to the small-to-medium-size website - it offers a great list of features and gives you *free unlimited bandwidth*. Really, with Cloudflare, the amount of money you save on bandwidth is only limited by how much you can make HTTP-cacheable. Incredible!

In addition, Cloudflare's recent HTTP/2 upgrade means that cached resources will be served over HTTP/2 connections, gracefully downgrading to HTTP/1 where necessary.

I've also found Cloudflare easy to use and setup - its web interface is miles ahead of Amazon's, for example.

Cloudflare seems to have a great dedication to speed - they're frequently the first CDN to publicly deploy performance features, like HTTP/2. As of February 2016, they are the only CDN to walk out with a perfect score from istlsfastyet.com, making Cloudflare (theoretically) the fastest CDN for SSL-served content.

Cloudflare's "market advantage" is in the wide variety of features they offer - unfortunately, some of these are simple bloatware. For example, they can (if you turn these features on) inject some scripts into your site that do things like scramble email addresses or other trivial tasks you could do yourself. Also, some of the performance related features, such as RocketLoader, strike me as fancy proprietary junk - not likely to really improve performance in most situations.

Finally, Cloudfront can sometimes perform poorly on bandwidth tests. If serving large, ~100+ MB files, you should probably look elsewhere.

Amazon CloudFront

When looking for a CDN, many want to hitch themselves to a big, proven company. Amazon certainly fits the bill, and CloudFront is widely used for that reason.

However, it's a bit of a bear to use sometimes. The web interface isn't great, and the API-based tools are similarly difficult to understand. Invalidations - removing content from the cache - are a pain, and actually cost you money.

The API is extensive, making CloudFront a good choice for complex workflows. The list of PoP locations is long as well.

Windows Azure

Windows Azure offers their own CDN - interestingly, they recently partnered with Akamai, the 8-ton-gorilla in the space. Unfortunately, the exact details of this partnership are unclear - I would not count on your assets being served by Akamai's PoPs just yet.

Azure performs well in CDN comparison tests, especially when transferring large files.

Azure's bandwidth prices are comparable to Amazon CloudFront, though their "Premium" offering is nearly double the price.

Unfortunately, Azure does not allow SSL with custom domains.

CacheFly

An interesting vendor, CloudFly's PoP locations are broadly comparable to Amazon CloudFront, with the addition of some PoPs in Canada and even one in South Africa.

CacheFly is clearly trying to corner the market on beating their competitor's bandwidth numbers. Every benchmark I could find routinely put CacheFly at the top when large files were concerned.

Unfortunately, speed isn't cheap. Prices are 50-100% more than Amazon CloudFront.

Checklist for Your App

- **Use a CDN.** Simple as that - pick a vendor based on price and point-of-presence locations relative to your users.

Interacting with (SQL) Databases

The database can be a scary thing. Bigger companies will almost always employ a DBA - a database administrator - whose *sole job* is to be the person responsible for most of the things I'm going to cover in this lesson. I'm not a DBA, but I do know some things about interacting with SQL databases!

This article is mainly going to talk about Postgres, the most popular SQL database used in Rails applications. Most of it, however, is broadly applicable to all SQL databases.

Indexing and You

What happens when you look for a `User` with an email of `donaldtrump@gmail.com` ?

Usually, a database will do what's called a *sequential scan*: it simply looks at each and every row in the database and compares the row's `email` field to your search. However, by adding an *index* to the `email` column, we can do an *index scan* instead. Unlike a sequential scan, index scans are far faster - instead of searching in linear time, we can search the database in logarithmic time.

We can add indexes in our migrations:

```
add_index :users, :email
```

Why not just index all the things? Maintaining indexes is hard work - every time we add a new row (or update an existing email), we also have to update an index. Essentially, indexes trade write speed for read speed.

We can combine fields in our indexes too - the fields used in our index must exactly match the fields used in our query. For example, if we frequently query on `User` `email` and `name` , like so:

```
User.where(name: "Donald", email: "donaldtrump@gmail.com")
```

...then this index setup will work:

```
add_index :users, :email
add_index :users, :name
```

In this case, Postgres will combine the indexes into what's called a "bitmap index scan". However, we can also combine these indexes into one for a super-fast index:

```
add_index :users, [:email, :name]
```

Though normally we want to be pretty stingy with adding indexes, there are a few scenarios where you should almost always add an index:

- **Foreign keys** For example, if Users have many Posts, you'll want to index `user_id` on the Post model. Foreign key columns are guaranteed to be queried on frequently, so it makes sense to make them as fast as possible.
- **Polymorphic relationships** Polymorphic associations are another great place for indexes. If you have a generic Comment model that can be attached to Posts and Pictures, for example, make sure that you've got a combined index for `commentable_type` and `commentable_id`.
- **Primary keys** Postgres automatically creates a unique index on our `id` columns, so we don't have to do this ourselves. Double-check to make sure your database does, too.
- **updated_at** In Russian Doll caching schemes, you will probably frequently be querying on `updated_at` to bust caches.

Indexes have an order - usually, they'll be sorted in ascending order. Sometimes, this isn't appropriate - for example, you may frequently be querying based on `updated_at` if you're using a key-based expiration approach:

```
<%= cache [@product_group, @product_group.products.max(&:updated_at)] do %>
```

In this case, an ascending index is inappropriate - we probably want a descending index to make that "MAX" query as fast as possible.

```
add_index :product_groups, :updated_at, order: { updated_at: "DESC NULLS LAST" }
```

We can also do what's called a "partial" index - only indexing under certain conditions. This makes sense when we frequently query for only certain parameters. For example, if you frequently look up which customers have been billed, but never look up customers

that *have* been billed:

```
add_index :customers, :billed, where: "billed = false"
```

Also, we can index with expressions. A common case for an expression in an index is for user emails - frequently, we want to do a case insensitive search for emails and do a query that's something like `SELECT * FROM users WHERE lower(email) = "donalddtrump@gmail"`. We can create an index for this exact case:

```
add_index :users, :email, where: "lower(email)"
```

It's also worth noting that indexes should be unique indexes where possible - unique indexes help ensure your data matches your constraints, but they're also faster than regular indexes.

How to EXPLAIN ANALYZE

Postgres comes with an `EXPLAIN ANALYZE` query, which can be prepended to any query to show you how Postgres' query planner decides how to perform the query.

Did we mention Postgres has a query planner? Deciding *exactly* how to execute any given query is not an entirely straightforward decision for a database - it has to decide on thousands of different ways it could possibly join or execute even the simplest of queries!

Here's some example output:

```
EXPLAIN ANALYZE SELECT "rubygems".* FROM "rubygems";

QUERY PLAN
-----
Seq Scan on rubygems (cost=0.00..2303.32 rows=119632 width=47) (actual time=0.006..18.498 rows=119632 loops=1)
Planning time: 0.050 ms
Execution time: 25.286 ms
(3 rows)
```

Postgres believes this query will return 119k rows, and that each row is approximately 47 bytes (`width`). The `cost` parameter is an abstract, relative representation of how long it should take to execute something - what it's saying here is that it costs about "0"

to get the first row, and "2303.32" to get all the rows.

We also have the actual time required to run the sequential scan step here - 18.498 milliseconds.

99% of "I don't think my index is getting used?" problems can be solved by digging in to the "EXPLAIN ANALYZE" results. This is the primary reason I use `EXPLAIN ANALYZE` - so let's show an example of looking at indexes:

```
EXPLAIN ANALYZE SELECT "rubygems".* FROM "rubygems" ORDER BY "name";
```

```
Index Scan using index_rubygems_on_name on rubygems (cost=0.42..8921.50 rows=119632 width=47) (actual time=0.505..199.385 rows=119632 loops=1)
Planning time: 2.580 ms
Execution time: 206.392 ms
(3 rows)
```

Neat - you can see that this particular query uses a named index, and that this index takes about 200 milliseconds to complete the query. Generally, I look for slow queries in my performance monitor - like New Relic - then I pop open a `psql` session on my production database (or a copy of it if I'm paranoid) and start running `EXPLAIN ANALYZE` to figure out what work can be done.

If you're still not satisfied and need more output, you can use `EXPLAIN (ANALYZE, BUFFERS, VERBOSE)` to show even more data about the query plan, like how much of the query was served by the database's caches:

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE) SELECT "rubygems".* FROM "rubygems"
ORDER BY "updated_at";
Sort (cost=16075.23..16374.31 rows=119632 width=47) (actual time=145.414..185.748 rows=119632 loops=1)
Output: id, name, created_at, updated_at, downloads, slug
Sort Key: rubygems.updated_at
Sort Method: external merge Disk: 6056kB
Buffers: shared hit=1107, temp read=759 written=759
-> Seq Scan on public.rubygems (cost=0.00..2303.32 rows=119632 width=47) (actual time=0.008..17.226 rows=119632 loops=1)
Output: id, name, created_at, updated_at, downloads, slug
Buffers: shared hit=1107
Planning time: 0.058 ms
Execution time: 201.463 ms
```

Cleaning Up After Yourself - Database Vacuuming

Some databases (Postgres and SQLite being the best examples) use a technology called MVCC (multiversion concurrency control) to provide concurrent access to the database even in situations where database rows may be locked for updates. Instead of just straight-up locking a record for updating, an MVCC database will create a copy of the row, marking it as "new data", and the "old data" will be discarded once the "new data" transaction has been completed and written to the database.

The problem is that these bits of "old data" are sometimes left behind and not cleaned up properly. This is what the `VACUUM` instruction is for!

Vacuums is important for two main reasons:

- It saves disk space. This "old data" can take up a significant amount of space on a write-heavy database.
- The Postgres query planner (discussed above) uses statistics that may be thrown off by too much "old data" laying around. Vacuuming can make these statistics more accurate and, thus, the query planner more efficient.

Postgres comes with an "autovacuum" function which is not on by default - if running your own database, make sure this is on. Heroku, for example, autovacums by default. The autovacuum daemon does *not* automatically give disk space back, though - to do that, you need to use the special `VACUUM FULL`, which needs an exclusive lock *on the entire database*.

Turn on autovacuuming and, when updating your database or otherwise taking it offline for maintenance, I suggest also running a `VACUUM FULL` to keep your database tidy and your query planner accurate. This is doubly important for long-lived applications or applications with lots of writes.

Connection Pools and Thread Math

Here's a quick lesson on Ruby web-app thread math.

Scaling a Ruby web application usually means more threads and more processes - but, usually, you may not be thinking about how these application instances are talking to shared resources in your architecture. For example - when you scale from 1 to 10 servers, how do they all handle coordinating with that single Redis server?

This kind of thing can happen in a lot of different places:

- Your database. ActiveRecord uses a connection pool to communicate with your database. When a thread needs to communicate to the database, it spins up a new connection in the connection pool. This pool can have a limited size - for example, if that pool size is 5, only up to 5 threads can talk to your database at once *per-process*. 5 servers running a single process with 5 threads each means a total of 25 possible database connections. 5 servers running Puma in "clustered" mode with 3 workers and 5 threads per worker means a total of $5 \times 3 \times 5 = 75$ possible connections. Your database has a maximum connection limit - you can see how simply "scale the servers!" could possibly overload your database with more connections than it could handle.
- Redis, memcache, and other key-value stores. As an example, Sidekiq has a connection pool when communicating with Redis. Manuel van Rijn has made an excellent calculator specifically for calculating connection pool sizes with Redis and Sidekiq: <http://manuel.manuelles.nl/sidekiq-heroku-redis-calc/>

Go and check your architecture right now - how many connections can your databases support? As an example, Heroku Postgres' "standard 0" tier offers 120 connections. The entry level Heroku Redis offering allows just 40 connections at once.

Now, how many connections "per server" are possible? Let's go back to a Sidekiq example. Let's say we're running Puma, with 2 workers and 3 threads per worker. Without a limit on the connection pool, Sidekiq will use up 6 connections per server at once - one for each thread on the server. With the entry level Heroku Redis plan, that gives us a theoretical limit of just ~ 6 servers (dynos in Heroku parlance) before we run out of connections. At that point, we'll start seeing dropped connections and other bad behavior.

Check your setup, and be aware for the next time you get crushed with load - you may have to start upgrading databases just to get more concurrent connections!

Disabling Durability

People frequently ask me how to speed up their test suite. There are a lot of ways to do this, but an easy one is to speed up your database.

In production, we want our database to be reliable and durable. SQL databases are designed around the ACID constraints - however, maintaining these guarantees of Atomicity, Consistency, Isolation and Durability is a costly endeavour. In a test

environment, we usually don't care about data corruption or loss. If it happens, we just run the test suite again!

What follows is a list of recommendations for settings to try to speed up your SQL databases in test environments. **Do not** apply these settings to production databases, or Bad Things will probably happen. Try these settings one at a time, and see if they speed up your overall suite time.

Of course, you probably shouldn't be writing to the database much during your tests anyway - but if you are (or you're stuck with someone else's application that does), here are those quick tips:

Place the database into RAMdisk

A "RAMdisk" is just a disk space partition that lives in your system's RAM rather than your disk drive. Treating your RAM like a disk drive can make for fast reads and writes - up to 10 times faster than an SSD.

Creating these is considerably easier on Linux systems, but most people develop locally on Mac, so my instructions will be for doing this on a Mac with OSX 10.11 (El Capitan). There are several good tutorials for running a RAMdisk database on Linux online.

First, we need to know how big our RAMdisk should be. The following SQL will print out the disk sizes of every database in our Postgres server:

```
SELECT pg_database.datname,  
       pg_size_pretty(pg_database_size(pg_database.datname)) AS size  
FROM pg_database;
```

| datname | size |
|-----------------------|---------|
| gemcutter_development | 1210 MB |
| gemcutter_test | 10 MB |

To be on the safe side, I'll create a 50MB RAMdisk for this application. Basically this involves using `hdiutil`, a system utility. The arguments are sort of complicated and hard to understand, so I use [a short bash script to format them correctly for me](#). Basically, it looks like this:

```

echo "Create ramdisk..."
RAMDISK_SIZE_MB=$2
RAMDISK_SECTORS=$((2048 * $RAMDISK_SIZE_MB))
DISK_ID=$(hdiutil attach -nomount ram://$RAMDISK_SECTORS)
echo "Disk ID is : " $DISK_ID
diskutil erasevolume HFS+ "ramdisk" ${DISK_ID}

```

Now I have a 50MB RAMdisk, mounted at `/Volumes/ramdisk`. We need to create what Postgres calls a "tablespace" at this disk location - we do that like this:

```
psql <database-name> c "create tablespace ramdisk location '/Volumes/ramdisk'"
```

Now we need our `database.yml` to run from the tablespace we just created:

```

test:
  adapter: postgresql
  ...
  tablespace: ramdisk

```

Run `rake db:test:prepare` again (because the database must be recreated) and voila - enjoy your free speed. In my testing with the Rubygems.org codebase, using a RAMdisk shaved about 10% of the total test suite execution time. Not a whole lot, but remember - I didn't have to change any application code, so I consider this win "free".

Turn off fsync and synchronous commit

Postgres (and many other SQL databases) maintains something called the "write-ahead-log". Postgres writes all modifications to the "write-ahead-log" *before* actually executing them - this allows the database to "start where it left off" if a catastrophic failures occurs during an operation.

Of course, in a test or even development environment, if this happens, well, we don't really care - so we can just turn it off.

We can't turn off the write-ahead-log entirely, but we can make it considerably faster.

You can disable writing the write-ahead log to the disk with the `fsync` configuration setting. `fsync` can be disabled in your `postgresql.conf` file or with the server command line.

In ordinary operation, the actual database transaction happens *after* the write-ahead-log has finished being written to. Again, since we don't care about the write-ahead-log, we can "de-sync" these operations by turning off `synchronous_commit` in our `postgresql.conf` for another speed boost.

Checklist For Your App

- **Get familiar with database indexing** Indexes are the key to fast queries. There are several situations where you should *always* be indexing your database columns - polymorphic associations, foreign keys, and `updated_at` and `created_at` if using those attributes in your caching scheme.
- **ANALYZE difficult/long queries** Are you unsure if a certain query is using an index? Take your top 5 worst queries from your performance monitor and plug them into an `EXPLAIN ANALYZE` query to debug them.
- **Make sure your database is being vacuumed** Autovacuum is mandatory for any MVCC database like Postgres. When updating or otherwise taking down a Postgres DB for maintenance, be sure to also run a `VACUUM FULL`.
- **Double check your thread math.** Make sure you have enough concurrent connections available across your application - do you have enough connections available at the database? What about your cache?
- **Consider disabling database durability in test environments.** Some, thought not all, test suites would benefit from a faster database. We can gain database performance by sacrificing some of the durability guarantees we need in production.

Is JRuby For Me?

When I got started with Ruby, in 2010, there really was no alternative to the default implementation of Ruby, CRuby. Most people were vaguely aware of JRuby, but it wasn't widely used, and making an application run on JRuby was a pain.

I'm happy to say that situation has changed - JRuby enjoys a vibrant contributor community, a wide following in enterprise deployments, and an increasingly bright future with skunkworks moonshots like JRuby+Truffle.

This lesson gives an overview of what the hubbub around JRuby is all about, where JRuby is going in the future, how to run your application on JRuby, and lists some practical tips for working and deploying on the Java Virtual Machine.

Why is JRuby fast?

Fundamentally, JRuby's philosophy is to re-use the decades of work that has gone into the Java Virtual Machine by re-using it to run Ruby instead of Java. There are a lot of JVM-based languages - you may have heard of Clojure, Scala, or Groovy. These language implementations turn source code into bytecode that the JVM can understand and run. Although we often speak of "the JVM", really, what we mean is the JVM *specification*. The JVM is an abstract concept, and there even many implementations of the JVM. The official one, which is the one JRuby installations use, is the HotSpot VM, distributed as a part of Oracle's Java Runtime Environment. HotSpot is over 15 years old, and a lot of work from large corporations has gone into making it as fast as possible.

In order to run your Ruby code as fast as possible JRuby makes a couple of tradeoffs:

- **JRuby uses more memory at boot.** Optimizing the performance of a language almost always involves a memory/CPU tradeoff. A language can be faster by eagerly loading code paths or caching the results of code execution, but this will necessarily result in higher memory use. JRuby, thus, tends to use more memory than the equivalent application running with MRI. There's a major exception, here though - JRuby tends to use more memory in the simplest case, though it uses far less memory than MRI as it scales. Also, this tradeoff of memory-for-performance is tuneable. We'll get to both points in a second.
- **JRuby takes longer to start up.** JRuby uses a limited amount of "just-in-time"

compilation, explained further below. However, when you restart a JVM, it does not save the optimized code snippets it generated, and has to re-generate them the next time it boots. The JIT, for this reason, may actually *slow down* really short tasks, such as installing a Rubygem.

- **JRuby has a warmup period.** Even after starting up, JRuby may be slower than CRuby. Several of JRuby's performance optimizations require code to be run a few times before JRuby can actually optimize it - for example, certain optimizations may only "turn on" once a bit of code is executed 50 or more times in quick succession.
- **No access to C extensions.** This used to be a much more painful tradeoff, but you can't use C extensions in JRuby. C extensions are essentially small C programs that interface directly with the Ruby VM, sharing memory space with it. This is pretty much impossible for JRuby. Some popular C extensions you may use are gems like `nokogiri`, which uses `libxml2` to use C to parse HTML.

In return, though, we get several benefits:

- **JRuby is fast after its warmup period.** In a long-running process, all of the start-up costs of JRuby get paid back. [JRuby maintains a set of running benchmarks at JRuby.org](http://jruby.org), and they show that, overall, JRuby is 1.3x as fast as Ruby 2.3.
- **True parallel threads, cheaply.** If a 30% speedup isn't that interesting to you (it is to me!), consider this - in JRuby, threads can be executed in parallel. There is no "interpreter lock" or "virtual machine lock" - threads can simply execute at the same time, making for true parallel execution. In highly concurrent situations (for example, serving many web requests at once), this makes the JVM much more efficient and performant than its CRuby cousin.
- **An extremely mature garbage collection process.** On the JVM, you actually have a *choice* of the garbage collector you use. In fact, you have 4 choices! Each one has more disadvantages and advantages than we can get into here, but know that garbage collection (and, therefore, overall memory usage) is far more mature and stable on the JVM than in CRuby.
- **Portability.** Java was designed to be a portable language. Officially, CRuby only supports Linux on x86 architectures. By contrast, the HotSpot VM that JRuby uses supports ARM and SPARC architectures, and has official support for Windows, Mac, Linux and Solaris.
- **Access to the Java ecosystem.** Although JRuby doesn't get C extensions, it does get access to running Java code inside of Ruby. Any Java library, of which there are *several*, can be called by your Ruby program.

This collection of tradeoffs means the JRuby makes a lot of sense in long-lived enterprise deployments, *especially* in situations where high concurrency is required (for example, maintaining many persistent connections at once with WebSockets). As an example, top-shelf game publishers have use JRuby and the Torquebox application server as the network back-ends to videogames for tracking achievements and user analytics.

Where is JRuby going?

Part of the reason why I wanted to cover JRuby in this course is because I feel its future is so bright. Development on the JRuby project has skyrocketed in the last few years, especially since the version 9.0.0.0 release, and it shows no signs of slowing.

Several performance optimizations are "on their way" in 2016.

JRuby has has an ace-in-the-hole when it comes to performance - JRuby+Truffle. This project is truly the dark horse of Ruby's future. Led by Chris Seaton of Oracle, its goal is to integrate JRuby with the Graal dynamic compiler and the Truffle AST interpreter.

The Truffle project is an *extremely* high performance implementation of Ruby - on JRuby's own benchmarks, it performs 64x faster than Ruby 2.3. If the Truffle project succeeds in its aims, it would be like the release of the V8 Javascript compiler all over again: an electric rebirth of an otherwise slow language.

However, the Truffle project is a long way to completely supporting Ruby. It only recently became complete enough to run a Sinatra application in 2015, and is only capable of "Hello World"-type applications in Rails as of 2016.

If you're interested in how Truffle works and why it's so much faster than CRuby, [check out Chris Seaton's blog](#). Chris is an *extremely* smart guy (he has a PhD in programming languages), and does a great job of explaining the extremely complex work that goes in to optimizing a language VM.

JRuby is slowly moving forward on compatibility and ease-of-use, though it's doing fairly well to begin with. Let's take a look at how difficult it is to convert an existing Rails application to JRuby.

Switching to JRuby

Running your Rails app on JRuby is a surprisingly simple process.

Installing JRuby

First, we have to make sure we're running an up-to-date version of Java. As of writing, this is Java 8. To check which version of Java you have, run `java -version` in the console. Weirdly, the Java Development Kit versions are prefixed with a "1", so we're looking for version "1.8.x":

```
$ java -version
java version "1.8.0_65"
Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)
```

If you've got an older version (`1.7.x` or `1.6.x`), you need to install a more recent version of the Java Development Kit.

Next, we have to install JRuby. This process will depend on the Ruby version manager you use (RVM, rbenv, etc.). I use `chruby` and `ruby-install`, so my install looks like this:

```
$ ruby-install jruby 9.0.5.0 && chruby jruby
```

You can test your JRuby install at this point by booting an interactive console with `jirb`. Just run `jirb` in your console.

Switching To JRuby-friendly Gems

Next, we need to make sure all of the dependencies of our project are JRuby-compatible. Here's the common ones:

- **Database gems need to be swapped for the JDBC equivalents.** `pg` must be swapped for `activerecord-jdbcpostgresql-adapter`, and so on.
- **Application servers should be changed for `puma`, `torquebox` or `trinidad`.** Application servers that `fork` won't work with JRuby.
- **`therubyracer` needs to be changed to `therhinoracer`** `therubyracer` uses the V8 engine to compile Javascript, and the `therhinoracer` uses the JVM.
- **Other C-extension gems that will fail to compile** - for example, `better_errors` depends on `binding_of_caller`, which uses a C-extension. `dalli`'s `kgio` also uses a C-extension. If you see gem fail to compile, it's got a C-extension.

We can make these gem swaps dependent on the version of Ruby we're running:

```

platforms :ruby do
  gem 'pg'
  gem 'unicorn'
  gem 'therubyracer'
end

platforms :jruby do
  gem 'activerecord-jdbc-adapter'
  gem 'puma'
  gem 'therhinoracer'
end

```

That's it!

For the most up-to-date information on this topic, I suggest reading Heroku's [step-by-step on moving an existing app to JRuby](#).

Practical tips

Java 7 shipped with a important feature called `invokedynamic` - it's a feature that increases the performance of dynamically typed languages (like Ruby!) running on the JVM. It's been supported in JRuby since version 1.1, and it's pretty critical for performance - applications can see a 2-4x speedup. Be sure to turn it on when testing out JRuby. You can enable `invokedynamic` on the command line: -

`Xcompile.invokedynamic=true` . This will significantly slow down startup time, especially for Rails applications, but should pay off after the JVM has warmed up.

Speaking of warmup and startup time, the JRuby developers realize that this is one of the major blockers for wider JRuby adoption. In JRuby 1.7, the `-dev` flag was introduced, which tells the JVM to basically prefer faster startup over increased optimization after boot. It turns off `invokedynamic` , as explained above, and disables the JVM bytecode compiler. Also, the JVM has a notion of running in "client" or "server" mode - these are actually two different compilers that are suited towards running a client app or a server app. Running with the `dev` flag enables "client" mode, further improving boot times. When developing, you may just want to turn this on all the time by changing the `JRUBY_OPTS` environment variable: `export JRUBY_OPTS="--dev"`

I mentioned above that JRuby tends to use more memory than CRuby. JRuby and the JVM, unlike CRuby, have the notion of a "maximum heap size". If we set a maximum heap size, JRuby *will not* use memory beyond that limit. The maximum can be set with

the `-Xmx` parameter: `-Xmx256m` sets the maximum heap size to 256MB. This could also be tuned upward in a production environment.

Most of the lesson material on profiling in this Guide does not apply to JRuby - however, the Java Virtual Machine has a mature ecosystem for profiling tools, all of which will work out of the box with JRuby. JRuby also ships with a profiler - it can be enabled with the `-profile` flag on the command line or used similarly to RubyProf:


```
require 'jruby/profiler'

result = JRuby::Profiler.profile do
  # code
end
```


If you've used the Spring preloader with Rails before, you might be wondering if a similar approach would work to offset JRuby's large startup costs. Spring uses the `fork` system command, however, which doesn't really make sense for use with the JVM. JRuby support is currently under development for Spring - [check here for more information](#).

If you're unsure about whether or not a particular gem will work with JRuby, check to see if the project already tests on JRuby. Most open-source projects use TravisCI or a similar service to run their tests for free, and you can see if their test suite passes on JRuby or not.

Build Jobs



✓ # 1055.1  `</>` Ruby: jruby-19mode

✓ # 1055.2  `</>` Ruby: jruby-19mode

You may notice that, while this Guide has a lesson on JRuby, it has no lesson on Rubinius, Opal or any other Ruby implementation. As far as I can tell, as of 2016, JRuby seems the only plausible alternative Ruby implementation for non-trivial production use. JRuby's wide adoption in the enterprise ensures that the community will remain strong for years to come, while other implementations have failed to gain major production deployments. JRuby's core team is strong too - with corporate support from RedHat and Oracle, JRuby has 14 committers with over 500 commits each. As a comparison, Rubinius has 7 (with many of them not contributing since 2009), and Opal just 3.

Checklist for Your App

- **Consider JRuby.** JRuby is a mature alternative to C Ruby, employed by many large enterprise deployments. It's become more usable with the JRuby 9.0.0.0 release, and development appears to only be speeding up as time goes on.

Alternative Memory Allocators

Have you talked to your memory allocator lately? Sent it a card for its birthday, or asked how its mother was doing? I didn't think so - memory allocators are often taken for granted by Ruby programmers. Just get me the memory I need, don't use too much, and do it quickly!

This is, of course, by design. Ruby is a language designed to shunt brain cycles *away* from memory allocation, type checking, and boilerplate and allow you to focus on code that is readable, beautiful, and fun to work with. If we wanted to think about how our memory gets allocated, we'd be writing C!

We've already discussed memory profiling and dealing with bloat and leaks. Most of those lessons have dealt with things going on *inside* of the Ruby VM. However, something has to be the liaison between the Ruby VM and the operating system's memory. That's what a memory allocator is for.

Your Rails application usually will (and frequently does) allocate memory *during a request* - this means that the performance of our memory allocator is an extremely important part of our overall application performance. In addition, memory allocators can have an impact on the overall memory usage of our programs, as they all deal with freeing and defragmenting memory differently.

One of the reasons I'm so interested in memory allocator choice for Ruby programs is that it's, possibly, a *free* improvement to your application's performance and memory usage. Changing memory allocators does not require any change to your application code, and is generally a completely painless process. I'm always for free wins - so let's dig in.

Allocating an object in Ruby

A quick warning - Ruby's garbage collector was under active development from versions 2.0 to 2.3. This lesson will refer to GC behavior in Ruby 2.3, which may differ from your version. For example, [many keys in `gc.stat` changed in version 2.1](#).

When you start a new Ruby process, Ruby automatically allocates a certain amount of memory for itself.

In Ruby, memory is organized into pages and slots - pages have many slots, and each slot contains a single `RVALUE`. These `RVALUE` objects normally has a size of 40 bytes. You can verify this yourself by checking `GC::INTERNAL_CONSTANTS[:RVALUE_SIZE]`.

Ruby starts up with a certain number of initial heap slots, determined by the environment variable `RUBY_GC_HEAP_INIT_SLOTS`. Roughly, the amount of memory your new Ruby process should take up should be equal to the number of heap slots it starts with, multiplied by the size of a heap slot (again, 40 bytes).

Once Ruby runs out of free heap slots, it asks the memory allocator to request more memory from the operating system for more heap slots. It doesn't request memory from the operating system every time you allocate an object - for example:

```
MyObject.new
```

...doesn't necessarily trigger a memory allocation.

However, if we were out of heap slots, Ruby *will* start up the memory allocator and ask for more memory. The number of heap slots it allocates depends on the GC variable `RUBY_GC_HEAP_GROWTH_FACTOR` - by default, this is 1.8. The number of total heap slots we'll want *after* allocating is equal to the current number of slots multiplied by this factor. If we've got 40000 full slots, and we want to grow the heap, we'll end up with 72,000 slots.

Enter malloc!

This process of enlarging the Ruby heap is managed by `malloc(3)`, a function in the C programming language that allocates memory dynamically. This function is actually carried out by any number of possible memory allocation libraries - by default, Ruby uses the standard `malloc` implementation included in `glibc`.

There are several alternative `malloc(3)` compatible memory allocators out there:

- `ptmalloc2` - This is `glibc`'s default `malloc`.
- `dlmalloc` - Doug Lea's Memory Allocator. `ptmalloc2` is a fork of this allocator, because `dlmalloc` has some critical limitations - it doesn't work with virtual memory and isn't designed to work with threaded programs. It was written in 1987, originally - back then, you only had 1 thread!
- `jemalloc` - Developed by facebook, `jemalloc` is designed to be performant for multithreaded programs.
- `tcmalloc` - "Thread-Caching Malloc", also designed for multithreaded work.

Developed by Google.

- `hoard` - Non-free (in the GNU sense of the word) allocator. Intended to improve fragmentation in multi-thread/multi-core programs.

Memory allocators have a lot of critical problems to deal with - the most important for us are performance, dealing with threads and reducing fragmentation.

What makes a memory allocator fast? Like most programs, the program does the least amount of work possible will be the fastest. All memory allocators are written in highly optimized C code - generally, their speed differences come in how efficiently they deal with problem of multithreading.

What makes a memory allocator good for multi-threaded programs? In general, the problem of managing threads that want to allocate memory at the same time is complex - we need to be sure that two threads don't claim the same piece of memory. If they do, that will almost certainly result in a crash of the entire program - quite literally a segmentation fault. Some memory allocators manage this problem with locks, like your database. Some allocators, like `tcmalloc`, use caching strategies to create almost-lockless access to memory for multiple threads. Other memory allocators that are particularly good for multithreaded programs use completely lockless designs.

What makes a memory allocator good at reducing fragmentation? In general, there is a tradeoff between memory usage and performance. Fast algorithms tend to use more memory, and we can reduce memory usage by sacrificing some performance. This is also true in memory allocators - `jemalloc`, while still fast, implements several strategies to reduce memory fragmentation that should, overall, reduce the amount of memory usage of your program.

How do I use an alternate malloc implementation?

In general, the easiest way to load an alternative `malloc` implementation right now in Ruby 2.3 is to "trick" Ruby into thinking that your chosen `malloc` is the system default `malloc` - we do this by using the `LD_PRELOAD` environment variable in Linux, or `DYLD_INSERT_LIBRARIES` on Mac.

For example, to install and use `jemalloc` on Mac:

```
$ brew install jemalloc
$ DYLD_INSERT_LIBRARIES=/usr/local/Cellar/jemalloc/4.0.0/lib/libjemalloc.dylib rub
y -e "puts 'jemalloc'"
```

On a Linux box, use `LD_PRELOAD` :

```
$ LD_PRELOAD=/home/nate/src/jemalloc-4.0.0/lib/libjemalloc.so ruby -e "puts 'jema
lloc'"
```

In Ruby 2.3, you may now configure `jemalloc` as the default memory allocator, if `jemalloc` is installed - just `./configure --with-jemalloc`. This support doesn't exist for any other allocator - all others must use `DYLD_INSERT_LIBRARIES` or `LD_PRELOAD`.

Memory allocator benchmarks

Which memory allocator do you use? I've prepared some benchmarks and ran all of the competitors on my local machine - consider running these yourself, especially on your production hardware. I'm going to use Sam Saffron's "GC stress test", which essentially just allocates a billion strings. In addition, I'm going to try a 60-second "dynamic" benchmark against Rubygems.org - the benchmark involves taking 32 load-testing workers and requesting random Rubygems from the site's index.

Sam's "GC stress test" will give us an idea of the overall speed of the allocator, while our dynamic benchmark against Rubygems.org will give us a better idea of our performance in a real-world, multi-threaded and multi-process situation.

glibc

In the GC stress test, `glibc` took 5.9 seconds to complete the test, on average. It was the slowest allocator on this measure. It took up 185MB of RSS, which made it "average" on my machine on this test.

In the dynamic benchmark against Rubygems.org, `malloc` performed somewhat inconsistently. Once, I even saw `malloc`'s end total of RSS usage *double* compared to other implementations. In general, however, it did well - its memory usage at the end of the test was about ~180MB of RSS per worker.

Here's the abridged about from `siege`, the load testing tool I used for this test:

```

Transactions:          14343 hits
Response time:         0.16 secs
Longest transaction:   0.80
Shortest transaction:  0.02

```

jemalloc

```
DYLD_INSERT_LIBRARIES=/usr/local/Cellar/jemalloc/4.0.0/
```

I expected `jemalloc` to be the clear winner here, but it didn't perform quite as well as I thought it would. It's the preferred allocator of the Discourse project, whose tech lead Sam Saffron claims saves them 10% of their RSS usage.

In the GC stress test, `jemalloc` completed it in an average of 5.2 seconds, roughly 15% faster than `malloc`. However, the GC stress test is a slightly unrealistic scenario - it's really the "worst case scenario" for any allocator, so it's difficult to draw any big conclusions from this test.

In our more real-world Rubygems.org test, `jemalloc`

```

Transactions:          14869 hits
Response time:         0.14 secs
Longest transaction:   1.08
Shortest transaction:  0.02

```

tcmalloc

```
DYLD_INSERT_LIBRARIES=/usr/local/Cellar/gperftools/2.4/lib/libtcmalloc.dylib
```

`tcmalloc` is distributed as part of Google's `gperftools` package - to use it, you'll need to `brew install gperftools` and look for it there.

`tcmalloc` did well in the GC stress test, with a fast 5.1 second completion time, making it also 15% faster than the standard `malloc`.

It performed more or less equal to the default implementation in the real-world Rubygems.org benchmark, however:

```

Transactions:          14334 hits
Response time:         0.16 secs
Transaction rate:      47.90 trans/sec
Longest transaction:   1.17
Shortest transaction:  0.02

```

hoard

```
DYLD_INSERT_LIBRARIES=~/.Code/Hoard/src/libhoard.dylib
```

Hoard is not available on `homebrew`, probably due to the licensing - it does not use a free software license. You have to download and compile the source yourself, and if you use it on a commercial project, you need to pay for a license.

`hoard` performed more or less equal to `jemalloc` on all measures - its GC stress test results were almost identical, and the real-world benchmark was also more or less equal.

```
Transactions:           14213 hits
Response time:          0.17 secs
Transaction rate:       47.43 trans/sec
Longest transaction:    1.07
Shortest transaction:   0.02
```

Conclusions

Changing your memory allocator is a low-downside, possible-high-upside change to your Ruby application. All of the memory allocators tested here were highly stable under production loads and situations, so I think they're all "ready for production". In addition, because changing memory allocators requires no code changes, I think it's a performance optimization anyone should try, especially if they're struggling with memory bloat.

In real-world, production situations, I've found `jemalloc` to be a slightly more consistent performer than the default. Although my synthetic "real-world" benchmark didn't really show any significant differences between implementations, I still think that in production situations - where many users may be hitting many different routes at once - these memory allocators can make a small difference. And since it's trivial to change, why not?

However, as shown by the real-world `Rubygems.org` benchmark, the performance implications may be minimal. None of the allocators showed significant differences in total memory usage at boot time or even in total RSS usage at the end of our real-world benchmark.

If you'd like to try an alternative allocator on Heroku, you can [check out the jemalloc buildpack that I help maintain](#). It's a 10-second install with the new native "multi-buildpack" Heroku features, and has performed well in production for my clients.

Checklist for Your App

- **Try a different memory allocator.** `jemalloc` is a well-tested and proven alternative. It may have a small impact on total memory usage and performance.

Making SSL Fast and Secure

The web is waking up to security. Thanks to recent revelations about the extent of government surveillance and ever-more high-profile attacks on corporations and celebrities, we're starting to realize that we must encrypt everything. Also until recently, doing this - SSLizing everything under the sun - was difficult and expensive. Now, with services like LetsEncrypt.org and SSL termination offered by intermediate CDNs like Cloudflare, it's easy and even free to set up SSL on any website.

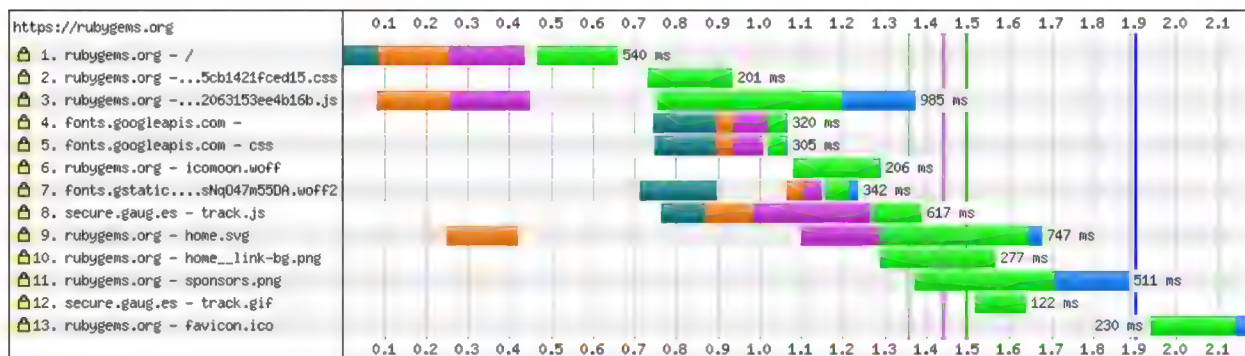
First, a little bit of definition - what's the difference between SSL, TLS, and HTTPS?

SSL and TLS are really the same thing - when SSL was invented by Netscape in the 90s, they called it the Secure Socket Layer. At the height of the internet boom in 1999, the Internet Engineering Taskforce (IETF) codified the until-then-proprietary technology as Transportation Layer Security, or TLS. Most people still call it "SSL", which is pretty much correct (since they're just different versions of the same technology), so that's what I'll use here to avoid confusion.

SSL sits between TCP and HTTP in the network stack - it lives on top of TCP, but beneath ("transporting") the application-layer concern of HTTP. HTTPS is literally just "HTTP over SSL", and is not a separate protocol to HTTP.

Also at the outset, let's get another fact straight - SSL does not impose enough of a performance cost to make any argument about "it's too slow to use." Cryptographic functions used by SSL are, with modern hardware, a negligible fraction of CPU time on client and server. SSL does not impact network bandwidth/throughput in any meaningful way. It does, however, impose a small latency penalty. This latency penalty is a necessary consequence of how SSL works, and we have a lot of knobs in the SSL protocol that we can twiddle to reduce this penalty.

As performance-minded web application developers, what we're concerned about is this:



This is a chart from webpagetest.org, charting Rubygems.org's initial page load time. SSL requires several network round-trips to establish, meaning that SSL will add (average round-trip time * # of round trips) of a delay to your initial page load. In the above case, SSL added 181 milliseconds - with careful configuration twiddling, we can reduce this by over 100 milliseconds, making SSL's overhead almost trivial.

It's worth noting that 181 milliseconds may not seem like a lot - it isn't, really. However, it does add up - SSL negotiation must happen on *every new TCP connection between client and server*, imposing huge penalties when several parallel connections are opened to download, say, your CSS, JS and images in parallel. Also, SSL hits mobile users the hardest - in connection environments that are spotty, SSL negotiation can take far longer as critical packets must be re-sent as they are dropped.

A Quick Overview of How SSL Works

First, we need to get on common ground regarding how SSL works. The average developer's knowledge of SSL probably only goes as deep as "it encrypts my HTTP connection so the NSA can't read it" - which is good, because SSL should be easy to implement for developers like us and shouldn't require an in-depth knowledge of the protocol.

SSL negotiation takes a few basic steps:

1. We have to open a new TCP connection. This requires a network roundtrip.
2. Next, the client (let's just say it's a browser), **sends a list of cipher suites** and other important information about its SSL capabilities to the server.
3. The server **picks a cipher from the list**, and **sends its SSL certificate back to the client**.
4. The client, which now has an agreed-upon common cipher and SSL version, **sends its key exchange parameters** back to the server.
5. The server **processes those key exchange parameters, verifies the message, and returns a 'Finished' message** back to the client.
6. The client **decrypts the Finished message**, and we now have a secure SSL connection.

This is pretty complicated. Security isn't simple. What you'll notice is that this process requires two full round-trips. With some work, we can reduce that to one in most cases, making our SSL experiences as fast as possible.

SSL Sessions

This is probably the most important SSL optimization we can make - session resumption.

With SSL sessions, the server can send a "session identifier" along with its certificate during the SSL negotiation process. Think of this SSL session identifier as a bit like the session cookie in Rails application. However, while a Rails session cookie represents a user login to your application, the SSL session identifier represents a set of agreed-upon SSL parameters. A session identifier basically just stands for "the client and server have agreed to this particular set of TLS version and cipher suite."

The advantage of this is that the client can store this session identifier. In the future, when opening a new connection, the client can *skip an entire network roundtrip*, because the cipher suite and TLS version have already been negotiated. This usually halves your total SSL negotiation time.

As an example, here's how SSL session resumption looks in `nginx.conf` :

```
http {  
    ssl_session_cache    shared:SSL:10m;  
    ssl_session_timeout 10m;  
}
```

A necessary consequence of SSL session resumption is that the server must keep track of these sessions - NGINX accomplishes the task with a cache. In the example above, sessions live for 10 minutes and the cache has a 10MB size. The need for a session cache can be alleviated by something called a session ticket - NGINX uses these by default. Check your server documentation for more details. Not all browsers support session tickets, either, so you'll probably still want a session cache.

SSL sessions are *not* enabled by default on most web servers. The reason is that it makes proper load balancing far more complicated - when enabling SSL sessions, consult your local DevOps guy and do some research on how it may affect your load-balancing.

OCSP Stapling

SSL certificates can be revoked. If, say, an attacker discovers your server's private key, they could pretend to be you and use your SSL certificate on their own servers. If this ever happened, you would want to revoke your SSL certificate.

Clients must verify whether or not the SSL certificate provided to them by the server has been revoked or not. Simply presenting a certificate to the client is not enough - the client has no idea whether or not this certificate is still good!

There are two ways for a client browser to do this - the Certificate Revocation List, and the Online Certificate Status Protocol.

The Certificate Revocation List, or CRL, is just an enormous list of SSL certificates that have been revoked, listed by serial number. This list is maintained by the Certificate Authority, the people you buy your SSL certificate from. The problem with the CRL approach is that this list is pretty long - downloading it and searching through it takes time. However, it can be cached by client browsers.

The Online Certificate Status Protocol, or OCSP, is the CRL's "real-time" equivalent. Instead of checking for the certificate's serial number in a huge list, the browser sends a network request to the certificate authority and asks "is this particular certificate #123 still valid?". To use a Rails metaphor: if CRL is `CertificatesController#index`, think of OCSP as `CertificatesController#show`. The disadvantage is that OCSP incurs an additional network round-trip, and OCSP's performance is *vastly* dependent on how fast the Certificate Authority's servers are. I hope you didn't pick a cut-rate discount CA!

What's a developer to do with this information? We can enable something called **OCSP stapling** - the server can include the OCSP response from the certificate authority when it presents its certificate to the client. In effect the server is saying, "you don't need to check if this revoked, here's a signed response from my Certificate Authority saying its valid".

Check your webserver's documentation to see if OCSP stapling is supported. Qualys' SSL test, explained in more detail below, will tell you if OCSP stapling is already enabled on your site.

It's important to note that OCSP stapling will only help for a limited subset of browsers - unfortunately, exactly *how* browsers check for certificate revocation varies *wildly* across browsers. OCSP stapling can't *hurt* the performance of any of them though, so it's "better safe than sorry".

HSTS

HSTS stands for HTTP Strict Transport Security. It's primarily a security measure - it protects against cookie hijacking and protocol downgrades. For example, if your users sign up for your site over SSL and receive a session cookie over SSL, that cookie is, so far, a secret and not decryptable by any men-in-the-middle. However, let's say that the next day the user types "<http://yoursite.com/>" into the address bar. The session cookie they received yesterday gets sent, in plaintext, across the network. Eventually, they'll probably get redirected to the SSL version of your website, but their session cookie has now been compromised and could be grabbed by men-in-the-middle looking to impersonate them. This is similar to how [Ashton Kutcher's Twitter account got hacked a few years back](#)..

HSTS closes these loopholes by telling the browser that it should *never* attempt to connect to your domain over an unencrypted connection. By turning on HSTS, you're telling any client browser that connects to you that the *only* way to connect to `yoursite.com` is over SSL. This prevents browser from accidentally presenting sensitive information in plaintext.

You should probably enable HSTS for the security benefits alone. It also has a nice performance benefit, though - it eliminates unnecessary HTTP-to-HTTPS redirects. Unless, for some weird reason, people must connect to your domain over unencrypted connections, you should turn on HSTS.

Cipher Suite Selection

Cipher suites can make a difference in SSL performance. Some cryptographic methods are just faster than others, so we should not support slow cryptographic methods on our servers.

Some SSL optimization guides may advocate this, and provide a list of "fast" cipher suites for your particular webserver. **I do not recommend choosing cipher suites based on performance characteristics**, however. Cipher selection is probably one of the most important parts of a secure SSL connection, because attacks against these ciphers are evolving all the time. Decisions on cipher suites should be made primarily with security in mind, not performance. Besides, the server always has the final choice as to what cipher will be used. During SSL negotiation, the client simply presents a list of ciphers it supports. The *server* is the member of the negotiation that decides exactly which cipher will be used, which means that most servers will intelligently pick the fastest and most secure cipher the client supports.

Instead, you should choose your server's list of supported ciphers based on [Mozilla's recommendations](#). Mozilla maintains a public wiki with up-to-date suggestions for secure and fast ciphersuites. Mostly, the decision depends on what level of browser support is required.

False Start

SSL sessions reduce our SSL negotiation by a full network round-trip for returning visitors; SSL False Start can help us reduce SSL negotiation by a full round-trip even for new visitors.

In the usual SSL negotiation, the client waits for an encrypted "Finish" message from the server before sending any application data. This makes sense - by that point in the SSL negotiation, the client and server have already agreed upon a cipher suite and TLS version along with the shared encryption key. Assuming no one has tampered with the negotiation process so far, the client can proceed. This eliminates another full network round-trip.

Support for false start depends largely on the client browser. IE will *a/ways* attempt false starts, Safari will attempt it if the ciphersuite enables forward secrecy, and Chrome and Firefox need forward secrecy and something called an "ALPN" advertisement.

Forward Secrecy

Forward secrecy is a property of some ciphers that essentially protects communications in the case of the private key being compromised. If the private key is discovered, the attacker cannot decrypt communications from the *past* that were transmitted using that certificate/private key.

Think about how important this is in protecting communications from a large, organized attacker. If a large organization, such as an ISP or a government agency, was recording *all* encrypted Internet traffic across the backbone, it could, in theory, decrypt those communications at a later date if they discover (or subpoena!) the private keys for the SSL certificate used. Forward secrecy prevents this.

Forward secrecy is required by Chrome, Safari, and Firefox to enable SSL false start.

If your cipher suites support Diffie-Hellman key exchange, you support forward secrecy.

ALPN

Chrome and Firefox additionally require Application Layer Protocol Negotiation (ALPN) in order to attempt SSL False Start.

Essentially, in the usual SSL handshake, the protocol of the *application layer* has not yet been negotiated. We've established our SSL tunnel, but we haven't decided things like which version of HTTP we'll use (HTTP 1.x, SPDY, HTTP 2, etc). ALPN allows servers to negotiate the application protocol *while they are negotiating the SSL connection*. That this would be a requirement for sending application *early* (that's all false start is) makes a lot of sense.

Most webservers work with ALPN, so long as your webserver is compiled against OpenSSL versions 1.02 or greater.

CDNs allow termination close to the user

As mentioned in the article about content delivery networks, CDNs allow connections to terminate close to your user's physical location. This can *greatly* reduce round-trip times during SSL negotiation, meaning CDNs make SSL even faster!

Check your certificate chain

Your webserver doesn't usually provide just a single SSL certificate to the client - it provides a chain of several certificates. Usually, this is your server certificate and an intermediate certificate.

Browsers ship with a small set of root certificates - these root certificates have been evaluated as trustworthy by the browser. When you buy an SSL certificate, your server certificate gets associated with one of these root certificates via an intermediate certificate.

In order to verify your server's SSL certificate, the browser must verify that the entire chain is correct. To this end, make sure that you're providing the intermediate certificates to the browser - otherwise, the browser will have to go and download the intermediates themselves, causing additional network round-trips.

Also, since the browser ships its own root certificates, your server should not provide the root certificate in the certificate bundle - it's completely unnecessary and a waste of bits.

To check what certificates you're providing, use Qualys' SSL tool (linked below).

Turn off compression

While you're digging through your server configurations, you may come across something called "SSL compression". In theory, this sounds great, right? Normally we're always looking to compress our responses. Not in this case, though.

Enabling SSL compression makes you vulnerable to the "CRIME attack", exposing you to session hijacking. In addition, SSL compression may attempt to re-compress already compressed assets, like images, wasting CPU. Most browsers disable SSL compression already, but don't turn it on for your server either.

Tools

This lesson has given you an overview of how to improve SSL negotiation times on your application. To that end, there are two excellent tools and resources you should be aware of:

- **The Mozilla Wiki and SSL Configuration Tool.** [The Mozilla Wiki](#) contains a lot of information about the technical implementation and background of the features outlined above. In addition, [Mozilla maintains a configuration-file generator](#) with their recommended SSL configurations for Apache, NGINX, Lighttpd, HAProxy and AWS Elastic Load Balancer.
- **The Qualys Labs SSL test** [Qualys maintains a tool for checking on the SSL configuration of any domain](#). Use this tool to double-check that you support all of the performance optimizations mentioned in this lesson.

Checklist For Your App

- **Test your SSL configuration for performance.** I prefer Qualys' SSL tool. The key settings to look for are *session resumption*, *OCSP stapling*, *HSTS*, *Diffie-Helman key exchange*, and *number of certificates provided*. Use Mozilla's configuration generator to get a set of sensible defaults.

The Easy Mode Stack

Developers are lazy. Sometimes, we just want the easy answer - that's why Stack Overflow (and copy-paste) is so popular. Sometimes, we can be intellectually lazy too. We'd just like someone more experienced than us to tell us what to do, or what tool to use, so we can get on with our day or whatever crazy user story the client requested.

However, sometimes "easy answers" can be an interesting starting point for better development. [Take Sandi Metz's rules](#):

1. Classes can be no longer than one hundred lines of code.
2. Methods can be no longer than five lines of code.
3. Pass no more than four parameters into a method. Hash options are parameters.
4. Controllers can instantiate only one object. Therefore, views can only know about one instance variable and views should only send messages to that object (`@object.collaborator.value` is not allowed).

They're not *really* rules, of course - they're more like guidelines. Sandi says "you should break these rules only if you have a really good reason or if your pair lets you." They're not infallible, and they're not true for all cases. Heck, they may not even be true for most cases, but *you should try to follow the rule first before breaking it*.

I often get asked whether or not I think technology A or approach B is good. I have compiled all of these opinions into this document, which I'll call my "Easy Mode Stack". This stack is intended to be a starting point in your own Ruby web application stacks - if you don't know what technology to use for any particular layer, just use the one I mention here. If you don't like it, fine, move on and try something else. But, I suggest you try the one listed here first.

The Easy Mode Stack represents a stack that I think is the best combination between cost, ease of use, and performance available today. Where I mention a particular vendor, I do not have any commercial relationships with that vendor.

Content Delivery Network: *Cloudflare*. I recommend Cloudflare because it's brain-dead simple to set up (change your DNS and kaboom, you're done) and free (with no bandwidth limits). However, I avoid most of Cloudflare's "add on" features, like Railgun and Rocket Loader. Be sure to turn on SSL and HTTP/2, if it isn't turned on already!

Reasons to deviate from this: If your customers are outside of the U.S., pay attention to point-of-presence locations and choose the CDN with the best performance and lowest latency for the geographical location of your customers.

Javascript Framework: *View-over-the-wire.* For new, greenfield apps, I recommend going with Turbolinks. For older, legacy apps, I recommend using [jquery-pjax](#). Turbolinks works much better with a "global" approach - the entire application should just be Turbolinks-enabled. jquery-pjax is much easier to sprinkle in here and there. Both technologies are fundamentally just HTML over AJAX. This approach is far simpler than 3rd-party Javascript frameworks such as React or Ember, and, as I discuss in the Turbolinks lesson, just as fast. *Reasons to deviate from this:* If you're already using a single-page-app framework, just stick with it. There is no good reason *not* to use either Turbolinks/view-over-the-wire or a single-page-app framework.

Webserver: *Nginx, with optional openresty.* Nginx seems to have emerged as the clear winner in the webserver wars, with significant memory savings and performance improvements over Apache. If you're interested in doing some heavyweight configuration with Nginx and would rather avoid its somewhat draconic config files, you can use [openresty](#) to script nginx using the Lua programming language. Neat! *Reasons to deviate from this:* h2o is an interesting new project that claims to be even faster than nginx.

Application Server: *Puma.* Puma combines an excellent I/O model with simple, easy-to-use configuration. Application servers are unlikely to be the bottlenecks in your stack, and Puma appears to be "fast enough". *Reasons to deviate from this:* Phusion Passenger Enterprise and Unicorn *behind* a reverse-proxy like nginx are also acceptable alternatives, but each comes with caveats. Passenger isn't free, and Unicorn won't run your application in multiple threads.

Host: *Heroku.* One thing I like about Heroku is that it forces you into some good performance best practices from the start - you must design to scale horizontally, rather than just "adding more memory!", and since the containers are so memory-constrained, you'll have to make sure your app isn't memory-bloated. It's worth noting that there is no performance difference between Heroku's 1x and 2x dynos. There may be a small boost in changing to a "PX" dyno, because those dynos are not on shared hosts, but such benefits will be marginal. *Reasons to deviate from this:* If your devops setup can't work on Heroku, you'll need to roll your own. Most people severely overestimate how much of a "special snowflake" their app is, however.

Webfonts: *Google Fonts*. With user-agent-specific optimization, a world-class CDN, and some excellent optimizations around the "unicode-range" property, Google Fonts delivers an enormous performance bang for, well, \$0. *Reasons to deviate from this:* If your designer needs a particular font, you'll have to host your own. Emulate Google's approach - CSS stylesheets with external font resources. Prefer WOFF2. Do not inline fonts. See the Webfonts lesson for more on what optimizations can be applied.

Ruby Web Framework: *Rails*. I've thought long and hard about this one, but I just don't see a use-case that Rails doesn't cover well. Its main competitors - Lotus, Sinatra, Cuba, and Volt - all suffer from the same flaws: they're equally as performant as Rails once they're on feature parity with Rails (see the "Slimming Rails" lesson) and none of them have the community or ecosystem Rails does. *Reasons to deviate from this:* There isn't a performance reason to prefer another web framework, though there may be aesthetic ones. If you don't believe Rails has made "the right choices" in terms of architecture or software design, I know I won't convince you otherwise.

HTTP library: *Typhoeus*. Typhoeus is really just a wrapper around `curl`. This is a good thing - it means Typhoeus is really good at making requests in parallel. Also, it's the *only* Ruby HTTP library I know of that doesn't use exceptions to note if a request has failed or not - see the Exceptions as Control Flow chapter for why this is important. Also, Typhoeus is the *only* Ruby HTTP library that comes with a built-in response cache. *Reasons to deviate from this:* If pluggability is important to you, use Faraday.

Database: *Postgres*. The NoSQL wars have cooled, and Postgres has come out on top. With full-text search, Postgres is also probably "enough" for 80% of applications that bolt-on heavyweight services like Elasticsearch. Also, if you really need to store JSON documents like a NoSQL database, [it turns Postgres is actually faster at that than MongoDB](#). *Reasons to deviate from this:* You are a DBA and know what you're doing. If you think set theory is a bunch of crap, use a NoSQL database.

Database Vendor: *Whichever is closest to you*. The amount of network latency between your application server and your database should be as low as possible - this connection will likely have dozens of roundtrips occurring *per request*. For this reason, it is absolutely imperative that your database and your application server be as physically close as possible. An easy way to ensure this is to just use the same vendor for your database as you do for your application hosting - Heroku Postgres for Heroku, Amazon RDS when using Amazon EC2, etc. *Reasons to deviate from this:* None.

Cache Backend: *Redis*. The popular choice here is Memcache, but, as shown in my caching benchmarks, it offers little, if any, performance advantage. Since I also recommend using Redis for your background job processor, simplify your stack and just use Redis for your cache backend as well. However, I would *not* recommend using the same Redis instance for both. When used for caching, [Redis must be configured for "least-recently-used" eviction](#), which is not the default. This eviction scheme is inappropriate for background jobs. *Reasons to deviate from this:* None.

Background Job Processor: *Sidekiq*. Consistently the fastest background job processor, and only getting faster - Sidekiq 4 was a near-order-of-magnitude improvement on previous versions. *Reasons to deviate from this:* If you need greater reliability and introspection, you should choose a database-backed queue. Currently, my favorite DB-backed queue is Que, discussed in the Background Jobs lesson.

Performance Monitoring: *New Relic*. If Skylight is good enough for you, then go for it - but I find its pricing scheme is just too much for most small applications. You can get a *lot* done with New Relic's free plan. Read the full lessons on each to make the decision for yourself. *Reasons to deviate from this:* AppNeta seems like a strong alternative. There is no reason *not* to have one of these tools installed.

Performance Testing: *Local, with siege, ab, or wrk*. All of these tools - `siege`, `ab`, and `wrk`, are local tools you can install anywhere. `siege` has an excellent feature that will hit URLs as listed from a file, and `wrk` is easily extensible with a Lua scripting engine. *Reasons to deviate from this:* There are a lot of 3rd-party vendors for this, discussed in the Performance Testing lesson. These vendors seem to only make sense if you want to integrate performance testing into a CI framework.

Real-time framework: *message_bus*. I cannot recommend ActionCable - at least not yet. WebSockets is simply overkill for most applications, and, as of Rails 5.0, ActionCable feels half-baked. `message_bus` uses polling, which should work for 80% of web applications, and achieves the same end result as ActionCable with far less complexity. *Reasons to deviate from this:* If you're really sold on ActionCable, go for it.

User Authentication *has_secure_password*. Sometimes I wonder if beginning Rails developers even know about this method, included in ActiveRecord. Using the secure BCrypt hashing mechanism, you can accomplish 80% of what most applications drop in Devise for. *Reasons to deviate from this:* You need OAuth integration. Don't do OAuth yourself.

Memory Allocator: *jemalloc*. As discussed in the lesson on memory allocators, most memory allocators can, at best, give you a tiny speed boost and *maybe* some 5-10% RSS savings. However, changing your memory allocator requires no code changes, and all of the allocators I've tested have been equally stable. *Reasons to deviate from this:* There's no good reason *not* to at least try an alternative allocator.

Ruby Implementation *CRuby*. CRuby continues to improve incrementally in terms of performance, though Matz has publicly announced his goal of a 3x speed improvement for Ruby 3. CRuby remains "fast enough" for most applications, and the drawbacks of JRuby - increased memory usage and startup time - make it still a bit of a pain. *Reasons to deviate from this:* If JRuby's developer-mode quirks don't bother you, go for it. It remains difficult to use CRuby for development and JRuby in production.

View Templates *erb*, or Slim if you must. *erb* templates remain 5-8x faster than HAML, and 2-4x faster than Slim. If, however, you *must* have a fancier view templating language, Slim is the fastest of all the alternatives. Slim even maintains a running benchmark that runs with their CI tests. *Reasons to deviate from this:* None.

The Checklist

1. **Ensure production application instance counts roughly conform to Little's Law.** Ensure your application instances conform to a reasonable ratio of what Little's Law says you need to serve your average load.
2. **95th percentile times should not be too extreme.** Across your application, 95th percentile times should be within a 4:1 ratio of the average time required for a particular controller endpoint.
3. **No controller endpoint's average response time should be more than 4 times the overall application's average response time.**
4. **Quantify the cost of an additional second of browser load time.** Post this number where your team can see it. Discuss the process of how you arrived at this number with your team and whoever makes the business decisions.
5. **Set a front-end load time budget, and agree on a method of measurement.** No, you won't be able to perfectly replicate an end-user experience - that's OK. Agree that load times exceeding this budget is a bug.
6. **Set a maximum acceptable response time and maximum acceptable 95th percentile time.**
7. **Set a page weight budget,** based on your audience's bandwidth and the other budgets you've set.
8. **Set up a long-term performance benchmark.** Run a benchmark on your site using tools like `siege` , `ab` , or `wrk` , or use a 3rd-party vendor.
9. **Learn to use profilers.** Use a profiler like `ruby-prof` to diagnose your application's startup time. Where does most time go during your app's initialization process?
10. **Perform an audit of your Gemfile with `derailed_benchmarks` .** Substitute or eliminate bloated dependencies - `derailed` 's "TOP" output should probably be 50-60 MB for the average app.
11. **Consider logging memory statistics in production.** Experiment with `ObjectSpace` by writing a logger for your application that tracks areas you suspect may be memory hotspots or use a pre-built logger like `gc_tracer` . If you're not logging memory usage over a week or month long timeframe, you're losing valuable data that could be used when tracking down memory leaks. Being able to track memory usage against deploy times is absolutely critical to avoid tons of hard, dirty debugging work.
12. **Set up `rack-mini-profiler` to run in production.** Use the optional `flamegraph` and `memory_profiler` add-ons. Use `rack-mini-profiler` to see how many SQL

- queries pages generate in your app. Are there pages that generate more than dozen queries or so, or generate several queries to the same table?
13. **Your application should be able to run in the production environment locally.** Set up your application so it can run in production mode locally, on your machine.
 14. **Developers should have access to production-like data.** Using production-like data in development ensures that developers experience the true performance of the application when working locally. For most apps, you can just load a sanitized dump of the production database.
 15. **Use a performance monitor in production** - NewRelic, Skylight, and AppNeta are all respected vendors in this space. It doesn't really matter *which* you use, just use one of them.
 16. **You should have only one remote JS file and one remote CSS file.** If you're using Rails, this is already done for you. Remember that every little marketing tool - Olark, Optimize.ly, etc etc - will try to inject scripts and stylesheets into the page, slowing it down. Remember that the cost of these tools is not free. However, there's no excuse for serving multiple CSS or JS files from your own domain. Having just one JS file and one CSS file eliminates network roundtrips - a major gain for users in high-latency network environments (international and mobile come to mind). In addition, multiple stylesheets cause layout thrashing.
 17. **Every script tag should have `async` and `defer` attributes. Do not script inject.** "Async" javascripts that download and inject their own scripts (like [Mixpanel's "async" script here](#)) are not truly "asynchronous". Using the `async` attribute on script tags will *always* yield a performance benefit. Note that the attribute has no effect on inline Javascript tags (tags without a `src` attribute), so you may need to drop things like Mixpanel's script into a remote file you host yourself (in Rails, you might put it into `application.js` for example) and then make sure that remote script has an `async` attribute. Using `async` on external scripts takes them off the blocking render path, so the page will render without waiting for these scripts to finish evaluating.
 18. **CSS goes before JavaScript.** If you *absolutely must* put external JS on your page and you can't use an `async` tag, external CSS must go first. External CSS doesn't block further processing of the page, unlike external JS. We want to send off all of our requests *before* we wait on remote JS to load.
 19. **Minimize Javascript usage where possible.** I don't care how small your JS is gzipped - any additional JS you add takes additional time for the browser to evaluate on *every page load*. While a browser may only need to *download* JavaScripts once, and can use a cached copy thereafter, it will need to *evaluate* all of that JavaScript on *every page load*. Don't believe me that this can slow your page

- down? Check out [The Verge](#) and look at how much time their pages spend executing JavaScript. Yowch.
20. **Use a front-end solution that re-uses the DOM, like Turbolinks or a single-page-app approach.** If you're on the "JavaScript frameworks are great!" gravy train, great - keep using React or Angular or whatever else you guys think is cool this week (wink!). However, if you're *not*, you should be using Turbolinks. There's just too much work to be done when navigating pages - throwing away the entire DOM is wasteful as events must be re-delegated and handlers reattached, Javascript VMs built and DOMs/CSSOMs reconstructed on every page load.
 21. **Specify content encoding with HTTP headers where possible.** Otherwise, do it with meta tags at the *very top* of the document.
 22. **If using `X-UA-Compatible` , put that as far up in the document as possible.**
 23. **`<meta name="viewport" ...>` tags should go right below any encoding tags.** They should *always* appear before *any* CSS.
 24. **Reduce the number of connections required to load a page.** Connections can be incurred by requesting resources from a new unique domain, or by requesting more than one resource at a time from a single domain on an HTTP/1.x protocol.
 25. **HTTP caching is great, but don't *rely* on any particular resource being cached.** 3rd-party CDNs for resources like JQuery, etc are probably not reliable enough to provide any real performance benefit.
 26. **Use resource hints - especially `preconnect` and `prefetch` .**
 27. **Be aware of the speed impact of partials.** Use profilers like `rack-mini-profiler` to determine their real impact, but partials are slow. Iterating over hundreds of them (for example, items in a collection) may be a source of slowdown. Cache aggressively.
 28. **Static assets should always be gzipped.** As for HTML documents, the benefit is less clear - if you're using a reverse proxy like NGINX that can do it for you quickly, go ahead and turn that on.
 29. **Eliminate redirects in performance-sensitive areas.** 301 redirects incur a full network round-trip - in performance sensitive code, such as simple Turbolinks responses, it may be worth it to render straight away rather than redirect to a different controller action. This does cause some code duplication.
 30. **Use a CDN - preferably one that supports HTTP/2.** Using Rails' `asset_host` config setting makes this extremely simple.
 31. **If using NGINX, Apache, or a similar reverse proxy, configure it to use HTTP/2.** NGINX supports HTTP/2 in version 1.9.5 or later. Apache's `mod_http2` is available in Apache 2.4.17 and later.
 32. **Most pages should have no more than a few thousand DOM elements.** If a

single page in your application has more than ~5,000 DOM elements, your selectors are going to be adversely affected and start to slow down. To count the number of elements on a page, use `document.getElementsByTagName('*').length`.

33. **Look for layout thrash with Chrome Timeline.** Load your pages with Chrome Timeline and look for the tiny red flags that denote layout thrashing.
34. **Experiment with splitting your application.js/application.css into 2 or 3 files.** Balance cacheability with the impact to initial page download time. Consider splitting files based on churn (for example, one file containing all the libraries and one containing all of your application code). If you're using an HTTP/2-enabled CDN for hosting your static assets, you can try splitting them even further.
35. **Double-check to make sure your site has sane cache control headers set.** Use Chrome's Developer Tools Network tab to see the cache control headers for all responses - it's an addable column.
36. **If running an API, ensure that clients have response caches.** Most Ruby HTTP libraries do not have response caches and will ignore any caching headers your API may be using. Faraday and Typhoeus are the only Ruby libraries that, as of writing (Feb 2016), have response caches.
37. **Make sure any user data is marked with Cache-Control: private.** In extreme cases, like passwords or other secure data, you may wish to use a `no-store` header to prevent it from being stored in any circumstance.
38. **If a controller endpoint receives many requests for infrequently changed data, use Rails' built-in HTTP caching methods.** Unfortunately, Rails' CSRF protection makes caching HTML documents almost impossible. If you are not using CSRF protection (for example, a sessionless API), consider using HTTP caching in your controllers to minimize work. See [ActionController::ConditionalGet](#)
39. **Use Oink or ps to look for large allocations in your app.** Ruby is greedy - when it uses memory, it doesn't usually give it back to the operating system if it needs less memory later. This means short spikes turn into permanent bloat.
40. **Audit your gemfile using derailed_benchmarks, looking for anything that require more than ~10MB of memory** Look to replace these bloated gems with lighter alternatives.
41. **Reset any GC parameters you may have tweaked when upgrading Ruby versions.** The garbage collector has changed significantly from Ruby 2.0 to 2.3. I recommend not using them at all, but if you must - unset them each time you upgrade before reapplying them to make sure they're actually improving the situation.
42. **Any instances of SomeActiveRecordModel.all.each should be replaced with SomeActiveRecordModel.find_each OR SomeActiveRecordModel.in_batches.** This

batches the records instead of loading them all at once - reducing memory bloat and heap size.

43. **Pay attention to your development logs to look for N+1 queries.** I prefer using the query-logging middleware shown in the lesson on ActiveRecord. `rack-mini-profiler` also works well for this purpose.
44. **Restrict query methods - where, find, etc - to scopes and controllers only.** Using query methods in model instance methods inevitably leads to N+1s.
45. **When a query is particularly slow, use select to only load the columns you need.** If a particularly large database query is slowing a page load down, use `select` to use only the columns you need for the view. This will decrease the number of objects allocated, speeding up the view and decreasing its memory impact.
46. **Don't eager load more than a few models at a time.** Eager loading for ActiveRecord queries is great, but increases the number of objects instantiated. If you're eager loading more than a few models, consider simplifying the view.
47. **Do mathematical calculations in the database.** Sums, averages and more can be calculated in the database. Don't iterate through ActiveRecord models to calculate data.
48. **Insertion, deletion and updating should be done in a single query where possible.** You don't need 10,000 queries to update 10,000 records. Investigate the `activerecord-import` gem.
49. **Background work when it depends on an external network request, need not be done immediately, or usually takes a long time to complete.**
50. **Background jobs should be idempotent - that is, running them twice shouldn't break anything.** If your job does something bad when it gets run twice, it isn't idempotent. Rather than relying on "uniqueness" hacks, use database locks to make sure work only happens when it's supposed to.
51. **Background jobs should be small - do one unit of work with a single job.** For example, rather than a single job operating on 10,000 records, you should be using 10,001 jobs: one to enqueue all of the jobs, and 10,000 additional jobs to do the work. Take advantage of the parallelization this affords - you're essentially doing small-scale distributed computing.
52. **Set aggressive timeouts.** It's better to fail fast than wait for a background job worker to get a response from a slow host.
53. **Background jobs should have failure handlers and raise red flags.** Consider what to do in case of failure - usually "try again" is good enough. If a job fails 30 times though, what happens? You should probably be receiving some kind of notification.

54. **Consider a SQL-database-backed queue if you need background job reliability. Use alternative datastores if you need speed.**
55. **Make sure external databases are in the same datacenter as your main application servers.** Latency adds up fast. Usually, in the US, everyone is in the Amazon us-east-1 datacenter, but that may not be the case. Use `ping` to double-check.
56. **Use a cache. Understand Rails' caching methods like the back of your hand.** There is no excuse for not using caching in a production application. Any Rails application that cares about performance should be using application-layer caching.
57. **Use key-based cache expiration over sweepers or observers.** Anything that manually expires a cache is too much work. Instead, use key-based "Russian Doll" expiration and rely on the cache's "Least-Recently-Used" eviction algorithms.
58. **Make sure your cache database is fast to read and write.** Use your logs to make sure that caches are fast. Switch providers until you find one with low latency and fast reads.
59. **Consider using an in-memory cache for simple, often-repeated operations.** For certain operations, you may find something like the in-memory `LRURedux` gem to be easier to use.
60. **Instead of requiring rails/all, require the parts of the framework you need.** You're almost certainly requiring code you don't need.
61. **Don't log to disk in production.** It's slow.
62. **If using Rails 5, and running an API server, use `config.api_only`.**
63. **Eliminate exceptions as flow control in your application.** Most exceptions should trigger a 500 error in your application - if a request that returns a 200 response is raising and rescuing exceptions along the way, you have problems. Use `rack-mini-profiler`'s exception-tracing functions to look for such controller actions.
64. **Use Puma, Unicorn-behind-NGINX or Phusion Passenger as your application server.** The I/O models of these app servers are most suited for Rails applications. If using Unicorn, it must be behind a reverse proxy like NGINX - do not use Unicorn in environments where you do not control the routing, such as Heroku.
65. **Where possible, use faster idioms.** See the entire Idioms lesson for commonly slow code that can be sped up by a significant amount. Don't go crazy with this one, though - always prefer more readable code over faster code, and allow your performance changes to be driven by benchmarks rather than speculation.
66. **Use streaming liberally with landing pages and complex controller endpoints.** Nearly every large website uses response streaming to improve end-user load times. It's most important to add "render stream: true" on landing pages and complex actions so that users can start receiving bits of your response as fast as

possible, reduce time-to-first-byte and allowing them to download linked assets in the `head` tag as soon as possible. You should also be streaming large file responses, such as large CSV or JSON objects.

67. **Use ActionController::Live before trying ActionCable or other "real time" frameworks.** If you don't need "real-time" communication *back* to the server, and only need to push "real-time" updates from server to client, Server Sent Events (SSEs) can be much simpler than using ActionCable. Consider polling, too - it is easier to implement for most sites, and has a far less complicated backend setup.
68. **Get familiar with database indexing** Indexes are the key to fast queries. There are several situations where you should *always* be indexing your database columns - polymorphic associations, foreign keys, and `updated_at` and `created_at` if using those attributes in your caching scheme.
69. **ANALYZE difficult/long queries** Are you unsure if a certain query is using an index? Take your top 5 worst queries from your performance monitor and plug them into an EXPLAIN ANALYZE query to debug them.
70. **Make sure your database is being vacuumed** Autovacuum is mandatory for any MVCC database like Postgres. When updating or otherwise taking down a Postgres DB for maintenance, be sure to also run a `VACUUM FULL`.
71. **Double check your thread math.** Make sure you have enough concurrent connections available across your application - do you have enough connections available at the database? What about your cache?
72. **Consider disabling database durability in test environments.** Some, though not all, test suites would benefit from a faster database. We can gain database performance by sacrificing some of the durability guarantees we need in production.
73. **Consider JRuby.** JRuby is a mature alternative to C Ruby, employed by many large enterprise deployments. It's become more usable with the JRuby 9.0.0.0 release, and development appears to only be speeding up as time goes on.
74. **Try a different memory allocator.** `jemalloc` is a well-tested and proven alternative. It may have a small impact on total memory usage and performance.
75. **Test your SSL configuration for performance.** I prefer Qualys' SSL tool. The key settings to look for are *session resumption*, *OCSP stapling*, *HSTS*, *Diffie-Helman key exchange*, and *number of certificates provided*. Use Mozilla's configuration generator to get a set of sensible defaults.